Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Amélioration des performances d'un metarepository

Vanpé, Jeremy

Award date:
2020

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Université de Namur
Faculté d'informatique
Année académique 2019–2020

Amélioration des performances d'un metarepository

Jeremy Vanpé

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent Englebert

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

La question de la performance est un élément majeur dans la définition des exigences non fonctionnelles d'une application. Bien que ces exigences ne soient pas toujours définies de manière formelle, cet aspect n'en demeure pas moins important. En effet, de gros défauts de performance pourraient avoir des conséquences fâcheuses sur l'utilisabilité d'une application et en freiner l'adoption par les utilisateurs. Bien entendu, les outils MetaCASE par leur utilité pour l'industrie logiciel, n'échappe pas à la règle.

Partant du constat de l'existence de problèmes de performance dans la couche de persistance d'un MetaCASE, ce mémoire proposera une solution, pour répondre à cette problématique. Les avantages qu'apportent la solution en matière de performance seront objectivés, en suivant une approche méthodologique.

Abstract

The notion of performance is a major element in defining the non-functional requirements of an application. Although these requirements are not always formally defined, this aspect is nonetheless important. Indeed, major performance defects could have unfortunate consequences on the usability of an application and slow down its adoption by users. Of course, MetaCASE tools by their utility for the software industry, are no exception to the rule.

Based on the observation of the existence of performance issues in the persistence layer of a MetaCASE tools, this thesis will propose a solution to address this problem. The advantages of the solution in terms of performance will be objectified, following a methodological approach.

Remerciements

Je souhaitais tout d'abord remercier sincèrement le professeur Vincent Englebert qui été mon promoteur pour ce mémoire. Grâce à lui, j'ai eu le chance de mener un travail sur un sujet particulièrement inspirant. Il a su se montrer disponible. Chacune de nos conversations et chacune de ses remarques ont été d'une grande aide dans la réalisation de ce travail.

Je tenais également à adresser mes remerciements à l'ensemble des professeurs de l'UNamur et de l'EPFC qui ont jalonnés mon parcours académique. Chacun d'entre-eux a sans conteste participé à la construction de ma passion pour l'informatique et à mon esprit critique.

Je remercie aussi tout particulièrement ma compagne, qui m'a démontré à maintes reprises son soutien indéfectible et qui m'a permis grâce à chacun de ses encouragements et commentaires de réaliser ce travail. Enfin je tenais à remercier ma maman qui a sans nul doute contribué à l'homme que je suis aujourd'hui, et qui je l'espère serait fière de moi.

Table des matières

1 Introduction

La notion de performance est un élément important à prendre en compte dans le processus de développement logiciel. En effet, celle-ci représente une des exigences non fonctionnelles majeures d'une application. La définition des exigences en cette matière est souvent au coeur des réflexions durant la phase de conception de l'architecture d'une application. Cependant, les préoccupations concernant la performance ne sont pas limitées à la phase de conception des applications. Elles accompagnent généralement l'application dans bon nombre d'étapes de son cycle de vie. Il n'est d'ailleurs pas rare dans l'industrie de constater que les évolutions successives d'un logiciel, ont conduit à la dégradation de ses performances. Ces préoccupations affectent non seulement les applications traditionnelles (de gestion) mais également des applications plus complexes. Il n'est donc pas étonnant que la notion de performance soit importante pour les outils de support de l'approche CASE(*Computer Aided Software Engineering*).

Nous l'aurons compris, la nécessité de satisfaire un certain niveau d'exigence en matière de performance sera au coeur de ce travail. Partant de problèmes de performance identifiés dans la couche de persistance d'un outils de MetaCASE (MetaDone), il sera question de suivre une approche méthodologique afin de fournir une réponse adéquate à ces problèmes. Ceci consistera tout d'abord à exposer un ensemble de pistes de solution inspirées à la fois des recherches présentées dans la littérature mais également d'autres observations et réflexions. Il s'agira ensuite de mettre en oeuvre une des solutions parmi les plus prometteuses et de réaliser une évaluation comparative des performances de cette option par rapport à l'implémentation actuelle. Évaluation comparative qui devra permettre de tirer des conclusions sur la validité de l'approche choisie.

Ce travail suivra donc le cheminement suivant. Dans un premier temps, nous détaillerons l'ensemble des éléments établissant le contexte de la problématique à résoudre. Une fois ce contexte défini, la section 3 présentera l'état de l'art en matière d'amélioration de performance et de scalabilité relatif aux outils de *Model Driven Engineering*. À l'issue de cette section, il sera dressé une liste des perspectives envisageables pour l'amélioration du repository de MetaDone. Perspectives inspirées par les travaux présentés et certaines autres réflexions et observations.

La section suivante portant sur la mise en oeuvre d'une solution (section 4), détaillera l'option choisie. Nous y retrouverons également, la méthodologie suivie pour son implémentation ainsi que les caractéristiques de celle-ci. Il sera ensuite question, dans la section 5.3, de l'évaluation de la solution mise en oeuvre. Celle-ci présentera, l'approche méthodologique suivie, ainsi que les résultats obtenus.

Enfin, nous reviendrons sur les points essentiels développés dans ce travail (section 6). Il s'agira d'exposer une analyse critique des apports au domaine, de l'approche méthodologique suivie, et enfin des résultats. Cette section envisagera également les perspectives futures issues de ce travail.

2 Contexte

2.1 MetaDone

Avant de présenter brièvement l'outil MetaDone, il est important de présenter le contexte dans lequel celui-ci s'inscrit.

Le processus de développement logiciel est défini comme étant un ensemble structuré d'activités requises pour développer un logiciel. Il s'agit d'un processus complexe. C'est pourquoi, le domaine de l'ingénierie logiciel, s'attache à formaliser des méthodes qui permettent de définir, d'abstraire, d'affiner et de documenter un logiciel [Isa97].

Au coeur de ces méthodes, on retrouve la création de modèles. Un modèle est défini comme une représentation simplifiée d'un système, qui met l'accent sur les éléments essentiels et fait abstraction des éléments considérés comme non essentiels [PK88].

Les préoccupations en matière de réduction des coûts de développement, de standardisation des spécifications et de l'augmentation de la qualité des artefacts produits, ont été à l'origine de l'émergence des technologies CASE (*Computer Aided Software Engineering*)[CR88, BK91]. C'est-à-dire d'outils intégrant des méthodes de génie logiciel, afin de faciliter toutes les étapes du processus de développement logiciel.

Les outils CASE ont donc pour objectif de simplifier le développement logiciel, en automatisant de nombreuses tâches du processus de développement logiciel (relatives aux phases d'analyse, de design, d'implémentation, ...). Bien que ces outils offrent des gains en terme de productivité, de réduction du temps de développement et de qualité des logiciels [Qat09], leur inconvénient majeur est que bien souvent ils ne se basent que sur une seule méthodologie, ce qui impose à l'utilisateur de se limiter à la seule expressivité permise par l'outil [Isa97].

Les conséquences de cette limitation sont nombreuses. Parmi celles-ci, l'on peut citer l'impossibilité pour le développeur de représenter le domaine de manière adéquate, la difficulté pour les utilisateurs de comprendre les modèles ou de mapper les concepts à leur domaine d'application, ainsi que l'impossibilité de produire des artefacts (rapport, code, ...) [Met]. De plus aucune méthodologie ne peut prétendre être la panacée en matière de développement logiciel. En effet, les domaines traités peuvent parfois posséder des particularités tellement spécifiques, qu'il serait difficilement concevable de fournir une méthodologie générique répondant à tous leurs besoins.

Pour bénéficier des avantages de l'approche CASE, les entreprises possédant des domaines spécifiques, se retrouvaient alors dans l'obligation d'adapter un outil à la spécificité de leur domaine, ce qui pouvait se revéler assez onéreux (option seulement possible pour de grosses entreprises) [Isa97].

La réponse à cette problématique se trouve dans les outils MetaCASE. En effet, ceux-ci offrent la possibilité de capturer les spécifications d'une méthode, pour ensuite générer, de manière automatique, un outil CASE depuis ces spécifications [ESU97].

Le fonctionnement des outils MetaCASE est basé sur une architecture à trois niveaux en rajoutant une couche d'abstraction par rapport aux outils CASE. Le niveau le plus bas est similaire à celui des outils CASE il s'agit de la couche modèle (niveau de représentation du système). La couche intermédiaire est celle du modèle de la méthode, c'est-à-dire un méta-modèle. Un métamodèle comprend les concepts, les règles et les notations de schématisation d'une méthode donnée. C'est le niveau qui est fixe dans un outil CASE. Le niveau supplémentaire des outils MetaCASE, est le niveau de langage de métamodélisation, c'est-à-dire la couche qui permet de définir un métamodèle [Met].

Afin de mieux appréhender le fonctionnement et l'architecture des outils MetaCASE on peut illustrer les différents niveaux d'utilisation expliqués dans [Isa97] :

- Au niveau langage de métamodélisation, les développeurs de MetaCASE construisent les composants génériques de l'outil et décrivent la structure de base d'une ou plusieurs techniques de métamodélisation à utiliser pour capturer les spécifications d'une méthodologie.
- Au niveau métamodèle, les développeurs de CASE utilisent les techniques de métamodélisation fournies et les composants génériques pour créer un outil CASE personnalisé.
- Au niveau modèle, on retrouvent les développeurs qui utilisent les outils CASE personnalisés afin de développer des systèmes logiciels.

Il existe un certain nombre d'outil MetaCASE disponible sur le marché, parmi ceux-ci, on peut citer GME [LMB+01], MetaEdit+ [TR03] et enfin celui au coeur de ce travail MetaDone [Eng13]. MetaDone est donc un MetaCASE. Il a été développé par le PReCISE Research Center de l'Université de Namur et bénéficie de nombreuses contributions issues des activités de recherche et d'enseignement.

L'architecture de MetaDone est une architecture en trois : couche présentation, logique et repository. L'application est implémentée en Java en suivant le standard OSGI [Eng20]. OSGI (*Open Services Gateway initiative*) est un framework permettant de mettre en place une architecture modulaire pour les applications [Wik20b]. Les applications sont ainsi découpées en composants appelés *bundles*. Cette architecture permet d'obtenir des systèmes extensibles (mécanisme de plugins). Ce type d'architecture se retrouve notamment dans des applications comme des IDE (Eclipse[1] par exemple) ou dans des serveurs d'application Java (comme JBoss[2]).

[1] `http://www.eclipse.org/`
[2] `http://www.jboss.org/`

2.2 Le repository de MetaDone

Le repository est un élément central de MetaDone et est au coeur de ce travail. Il permet de stocker les artefacts créés afin de gérer des modèles et des métamodèles. Il se base sur le langage MetaL. Ce langage est conçu en deux couches distinctes : MetaL_1 et MetaL_2.

2.2.1 MetaL_1

Cette section est basée sur [Eng20].

MetaL_1 est un langage extrêment expressif possédant un très petit nombre de concepts. L'élément centrale de ce langage est la notion de *DataObject*. D'ailleurs, les seuls éléments gérés par le repository sont des *DataObjects*. Les *DataObjects* peuvent exercer des fonctions aussi appélées *Facets*. Il existe deux types de *Facets* principales : les *Objects* et les *Properties*. L'utilisation de la notion de *Facets* a été inspirée des travaux de [OS97].

Nous pouvons détailler les éléments du langage comme suit :

- **DataObject** : Ensemble d'éléments qui sont des *Objects* ou des *Properties* (voir les deux).
- **Object** : Ensemble d'artefacts possédant une identité propre.
- **ObjectType** : Sous-ensemble d'*Object*. Tout élément d'*Object* doit posséder au moins un type qui est un *ObjectType*. Il peut également en posséder plusieurs.
- **Property** : Ensemble d'artefacts représentant un lien entre deux *Objects*. Une instance est caractérisée par un *range* et un *domain*.
- **PropertyType** : Sous-ensemble de *Property*. Tout *PropertyType* possède un nom unique. Tout *Property* est typé par exactement un et un seul *PropertyType* que l'on appelle également son type. Un *PropertyType* peut-être le type de plusieurs *Properties*.
- **Object**X : Sous-ensemble d'*Object*. Où X représente l'un de ces suffixes : Integer, Boolean, String, Char, Float. Un *Object-X* est donc implicitement un *Object*, mais il possède une valeur dont le type est en accord avec le suffixe.
- **Type**X : Sous-ensemble singleton de *ObjectX*. Son unique instance représente le type X et il possède donc une valeur par défaut qui est respectivement 0, false, "", ” et 0.0.

Le langage défini également la relation *isa*, qui permet d'établir des relations entre les types d'*Objects*. Cette relation offre la possibilité de représenter une relation d'héritage ou de spécialisation.

Pour compléter les types présentés, le langage défini également un ensemble de règle pour ces types :

- Tout *DataObject* doit posséder au moins une *Facet*
- Toute instance d'un *ObjectType* A est également une instance de ses supertypes.
- Les *domain* et *range* d'un *PropertyType* doivent être des *ObjectTypes*
- Les extrémités d'un *Property* doivent être typées correctement par rapport aux extrémités de son modèle.
- Tous les *Objects* possédant une valeur sont distincts.
- Les *PropertyTypes* dont les domaines appartiennent à une même famille pour la relation *isa* ont des noms distincts.
- Si A est un sous-type de B et que B a pour type T alors A a également T comme type.

2.2.2 MetaL$_2$

Malgé l'expressivité importante de MetaL$_1$, il manque encore certains concepts nécessaires pour permettre de définir des métamodèles à partir de ce langage [EH07]. C'est pourquoi s'ajoute une couche d'abstraction supplémentaire MetaL$_2$. Ce niveau d'abstraction additionnel est entièrement défini en MetaL$_1$.

Ce langage de méta-métamodélisation défini quatres concepts principaux :

- **MetaObject** : Les *MetaObjects* sont déstinés à représenter les objets d'un métamodèle. Ils sont identifiés par un nom. Un *MetaObject* peut avoir des *MetaProperties* et peut également être connecté à d'autres *MetaObjects* par un *MetaRole*. Il est à noter que tout élément est MetaObject. En effet les concept de *MetaModel*, de *MetaProperty*, et de *MetaRole* sont des sous-types de *MetaObject*.
- **MetaProperty** : Un *MetaProperty* est lié à un *MetaObject*. Il possède un type (Integer, Boolean, String, Char, Float) ainsi qu'une cardinalité exprimée en entier.
- **MetaRole** : Un *MetaRole* représente une relation entre deux *MetaObjects* (un *domain* et un *range*). Il possède également une cardinalité.
- **MetaModel** : Un *MetaModel* quant à lui est composé d'un ensemble de *MetaObject*.

2.2.3 Metarepository

Après avoir introduit brièvement les deux niveaux du langage MetaL, il est important de mentionner que les couches du langage se retrouvent dans l'implémentation de MetaDone, chacune à des niveaux différents.

L'implémentation de MetaL_2 se trouve au niveau de la couche que nous appellerons business (situé dans le bundle metadone_business). Il s'agit de la couche directement manipulée par les concepteurs de metamodèles [EH07]. Au niveau d'abstraction juste en-dessous, nous retrouvons l'implémentation de MetaL_1. Il s'agit de la couche destinée à être manipulée dans le cadre de la persistance des données. Cette couche est composée d'un ensemble d'interfaces et de classes abstraites afin de permettre d'être implémentée par différents types de technologies de persistance[Eng20]. Un diagramme de classe de cette couche est présenté à la figure 1

La couche de persistance de MetaDone possèdent des particularités très différentes de celles des applications traditionnelles (de gestion). En effet, celle-ci manipule des objets qui définissent d'autres objets eux-même définissant d'autres objets pouvant être persistés et ainsi de suite (ce qui implique des relations réflexives). Cette couche de persistance se caractérise également par un assez petit nombre de concept manipulé (en l'occurrence quasiment un seul puisque tout est *DataObject* dans MetaL_1), associée à l'utilisation de relations réflexives, ce qui peut conduire à la création de très grands groupements d'objets (potentiellement infinis). Nous retrouvons, généralement, les mêmes caractéristiques dans tous les repositories qui gèrent des données de niveau *meta*, tels que ceux destinés aux autres outils de MetaCASE ou ceux destinés aux applications de Web sémantique (OWL *Ontology Web Language*) [CFB10].

Pour distinguer donc ce type de repository dans la suite de ce travail, il paraissait pertinent de définir un terme spécifique le caractérisant. Celui de metarepository a été choisi.

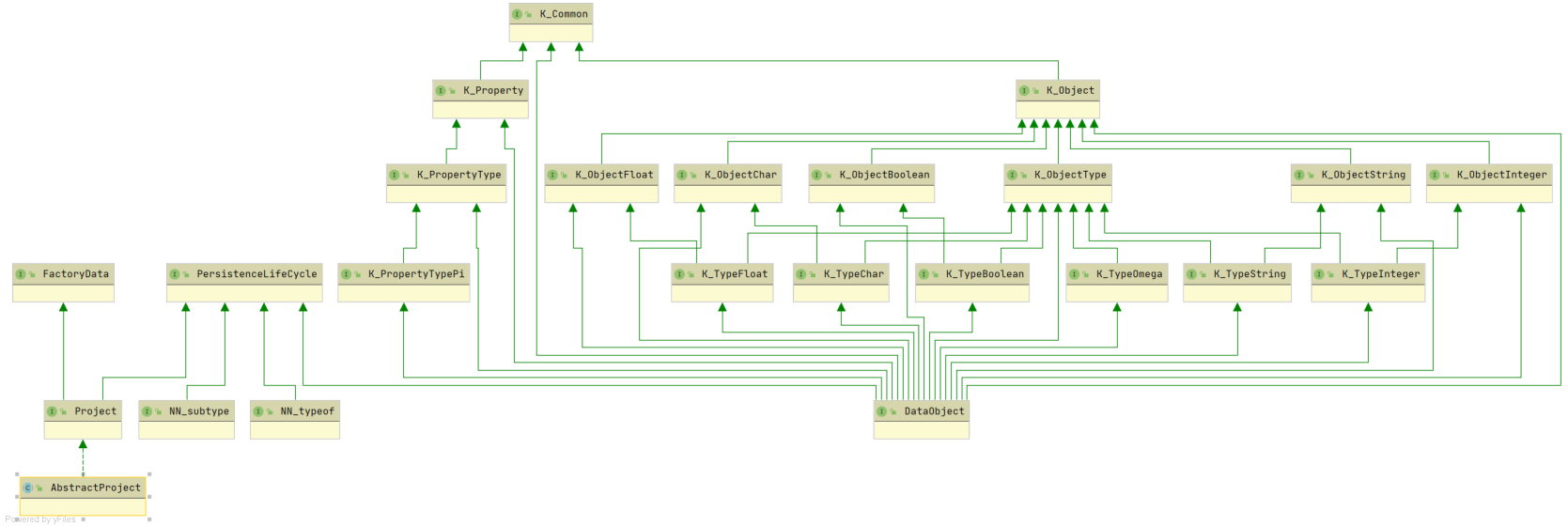

FIGURE 1 – Diagramme de classe de la couche d'abstraction du metarepository

2.3 Implémentation actuelle du metarepository

Comme expliqué dans la section 2.2.3, le metarepository possède une couche d'abstraction qui permet de découpler le code business du code d'accès aux données dépendant d'une technologie spécifique de persistance. Cette couche est donc composée d'un ensemble de classes abstraites et d'interfaces (situées dans le bundle repository), qui peut être implémentée par différents types de technologies de persistance.

L'implémentation actuelle du metarepository se base sur l'API standard JPA (*Java Persistence API*. Il s'agit d'une API permettant d'interagir avec une source de données relationnelles. Elle fournit un ensemble de classes, d'interfaces et d'annotations qui permet de mettre en oeuvre une technique d'*Object Relational Mapping* (ORM). C'est-à-dire de mapper un schéma relationnel à une représentation objet afin de simplifier sa manipulation dans une application orientée objet (réduction de l'*object-relational impedance mismatch* dont nous reparlerons plus en détail dans la section 4.1). Cette API repose sur l'utilisation d'annotation afin de réaliser le mapping des objets et de leurs variables d'instance vers une représentation relationnelle.

L'API JPA n'est qu'une couche abstraite. Pour implémenter la couche d'accès aux données d'une application, il est nécessaires d'utiliser une implémentation de ce standard. Il en existe plusieurs disponibles sur le marché (Hibernate[3], EclipseLink[4], ...). L'implémentation actuelle du metarepository utilise l'implémentation OpenJPA[5].

Pour le stockage des données, la base de données utilisée est une base de données H2[6]. Il s'agit d'une base de données open source, entièrement écrite en Java. Elle est utilisée par MetaDone en mode *embedded*. Le mode *embedded* consiste à ce que l'application ouvre la base de données dans la même *Java Virtual Machine*.

[3] `http://hibernate.org/`
[4] `http://www.eclipse.org/eclipselink/`
[5] `http://openjpa.apache.org/`
[6] `https://www.h2database.com/html/main.html`

2.4 Objectifs

Nous l'aurons compris MetaDone est un outil MetaCASE possédant un certain nombre d'avantages, non seulement d'un aspect fondation théorique (MetaL), mais également au niveau de son architecture logicielle (modularité, abstraction permettant de réimplémenter la couche de persistance, ...).

Cependant, il existe un certain nombre de problèmes de performance dans l'application MetaDone. Ces problèmes de performance s'illustrent lors de l'exécution d'action impliquant des opérations sur le metarepository. Comme par exemple lors du chargement d'un plugin qui implique la création ou l'accès à un métamodèle de taille importante, ou de manière plus générale sur la sauvegarde d'un travail en cours.

Bien que les exigences en matière de performance pour MetaDone ne soient pas explicitées de manière formelle par des scénarios d'attributs de qualité [BCK12], il semble évident que par exemple une action déclenchée par l'utilisateur prennant plusieurs secondes voir plusieurs dizaines de secondes, ne corresponde pas au standard actuel d'exigences en matière de performance. De plus, ces problèmes de performance affectent réellement l'expérience utilisateur, car bien souvent ces problèmes de performance se manifestent par des temps de latence dans l'interface de l'application (freeze) qui rendent diffficilement lisible le moment où l'utilisateur va pouvoir à nouveau interagir avec l'application (attribut de qualité d'utilisabilité).

Pour répondre à cette problématique, ce travail portera sur l'amélioration de la couche de persistance et d'accès aux données (metarepository) de MetaDone. Il s'agira donc de présenter les différentes possibilités d'amélioration des performances d'un metarepository, d'intégrer la où les pistes paraissant les plus prometteuses, et enfin d'utiliser une approche méthodologique afin d'évaluer les gains apportés par celles-ci.

3 État de l'art

3.1 Le Model Driven Engineering

Comme évoqué dans la section 2.1, l'approche CASE (*Computer Aided Software Engineering*) a fait naître un certain nombre d'outils permettant de faciliter et d'automatiser l'utilisation de méthodologies. L'émergence de ces outils a permis de populariser une approche méthodologique appelée *Model Driven Engineering* (MDE).

Le (MDE) est une méthodologie de développement logiciel dans laquelle, comme son nom l'indique, les modèles sont les éléments centraux. Dans cette approche, le processus de conception d'un logiciel consiste à capturer les détails importants d'un système au sein des modèles et d'appliquer sur ceux-ci une série de transformations afin de produire automatiquement des artefacts logiciels, tels que du code source ou de la documentation.

Afin de pouvoir concevoir, manipuler, transformer des modèles, ceux-ci doivent se conformer à leur métamodèle, on dit qu'un modèle est une instance de son métamodèle. Nous reviendrons sur le concept de métamodèle par la suite.

Le MDE possède de nombreux avantages par rapport aux méthodes traditionnelles d'ingénierie logicielle, telles qu'une productivité et une réutilisabilité accrues. Ces bénéfices ont conduit une partie de l'industrie du développement logiciel à s'intéresser à cette approche et aux outils qui le supportent.

Il existe un certain nombre d'outils de métamodélisation (et de modélisation) sur le marché (Microsoft DSL Tools, MetaEdit+, GME, ...) permettant à l'industrie de bénéficier des avantages du MDE [GTSC15]. Chacun de ces outils possède généralement son propre écosystème. Parmi ces outils ou frameworks, l'un des plus largement adopté et faisant l'objet de nombreuses recherches est l'Eclipse Modeling Framework (EMF) [Wik20a] et notamment dans un domaine particulièrement intéressant pour l'amélioration du metarepository. En effet, ce framework a fait l'objet de nombreuses recherches relatives aux problèmes de performance et de scalabilité. Il paraît donc intéressant de s'y attarder quelque peu.

Ce framework a été très fortement influencé par le standard OMG Meta-Object Facility (MOF) pour la représentation et la manipulation des métamodèles. En effet, ECore le métalangage (langage permettant de définir des métamodèles) fourni par EMF, peut être vu comme une implémentation simplifiée de MOF [Eng20].

Le MOF est structuré comme une architecture à 4 niveaux :

- M3 : méta-métamodèle
- M2 : métamodèle (ex : UML)
- M1 : modèle du système réel défini, dans un certain langage
- M0 : système réel, système modélisé

La couche M3 représente donc une abstraction permettant de modéliser des métamodèles. Ce langage est dit auto-descriptif, c'est-à-dire qu'il permet de se définir lui-même. Ce langage

défini un ensemble de concepts de base tels que entité/classe, relation/association, type de données, référence, package, etc.

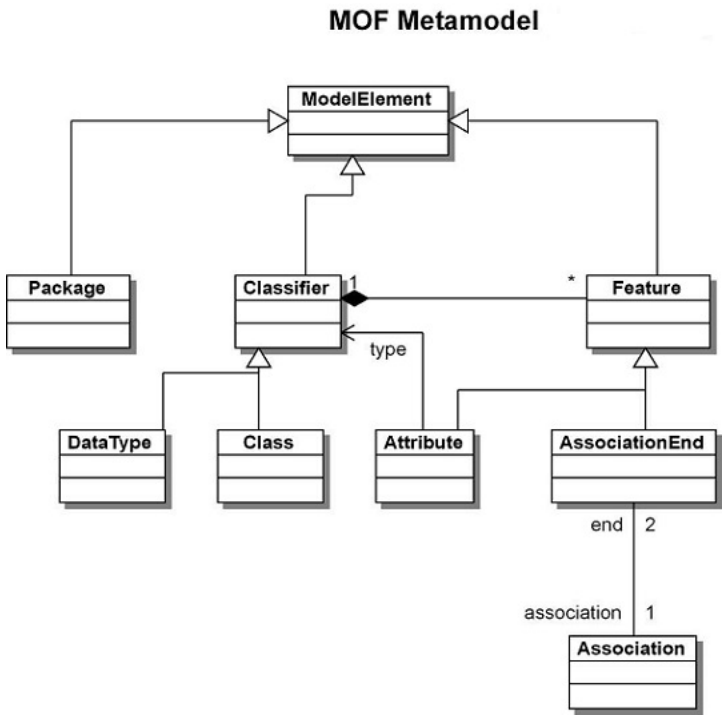

FIGURE 2 – Métamodèle MOF

Le modèle ECore quant à lui, peut être vu comme le langage de niveau M3 dans l'environnement EMF. Il définit donc un certain nombre d'éléments de base similaires à ceux du MOF tels que package (Epackage), classe (EClass), type de données (EDataType), attribut (EAttribute), opération (EOperation) et lien de référence (EReference).

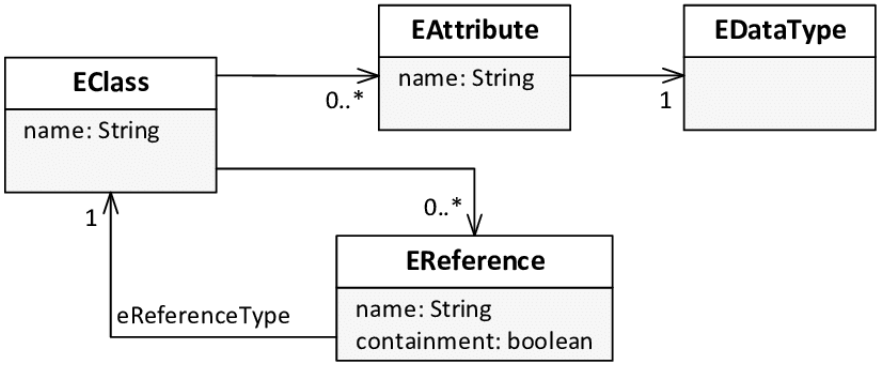

FIGURE 3 – Métamodèle simplifié de Ecore

Un autre standard OMG qui a été intégré dans EMF est le format de sérialisation des modèles sous forme de fichier XML, le *XML Metadata Interchange*, ce qui permet la persistance et le partage de modèles. Ce format est également supporté par de nombreux autres outils de gestion des modèles.

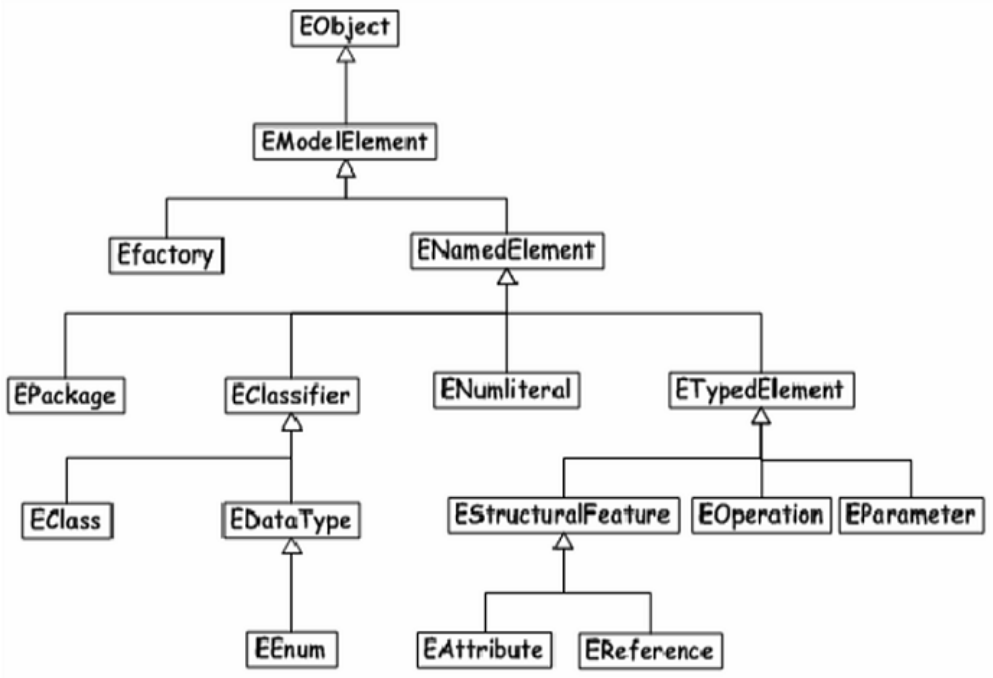

FIGURE 4 – Métamodèle Ecore

Cependant, même si par défaut EMF utilise XMI comme moyen de persistance des modèles, il a également comme avantage d'offrir un niveau d'abstraction entre la couche de persistance et le modèle (*model repository*). Grâce à cette abstraction, il est donc possible d'implémenter des solutions de persistance utilisant des technologies différentes sans devoir toucher au modèle. Cette approche un peu similaire à celle adoptée par MetaDone, a conduit de nombreux chercheurs à évaluer des implémentations alternatives du repository EMF (que nous aborderons dans la suite de cet état de l'art).

3.2 La problèmatique de scalabilité des outils MDE

Comme expliqué dans [BGS+14], l'émergence de nouvelles techniques et d'outils pour construire des systèmes complexes distribués a fait naître un besoin en matière de modernisation des systèmes existants.

Le processus de modernisation de système informatique doit suivre une approche systématique. Effectivement, il faut tout d'abord passer par une phase de création d'une abstraction à partir du code source par le biais du reverse engineering. L'abstraction ainsi produite permettra, par la suite, de comprendre, d'évaluer la qualité, de définir des architectures d'entreprises et finalement de pouvoir améliorer le système.

En fournissant une base commune afin de répondre à divers objectifs tels que le reverse engineering, la transformation de modèles et la génération de codes, l'utilisation des outils de *Model Driven Engineering*, comme EMF, semble être une approche naturelle pour répondre aux besoins de l'industrie en matière de modernisation logiciel.

Cependant, ces outils ont tout d'abord été conçus avec comme première intention de supporter des activités de modélisations essentiellement manuelles (modèles de l'ordre de quelques centaines d'éléments), mais ont montré certaines limites lorsqu'il est question de gérer des modèles de grandes tailles (de l'ordre de quelques milliers à quelques millions d'éléments), ce qui est généralement le cas de modèles générés automatiquement par des techniques de reverse engineering.

Nous parlerons, pour décrire ce problème, d'un manque de scalabilité. La scalabilité désigne dans ce cadre, la capacité d'un système à maintenir ses fonctionnalités et ses performances lors d'un changement d'ordre de grandeur de la demande (montée en charge) [Wik20c]. Étant donné qu'il existe une relation avec la notion de performance et que la scalabilité est également un élément important pour MetaDone, il semble intéressant, en vue de l'amélioration du metarepository de s'intéresser aux réponses apportées dans cette matière.

Ce manque de scalabilité des outils MDE représente donc un frein important à une large adoption dans l'industrie logiciel. C'est pourquoi ce sujet a fait l'objet de nombreuses recherches. Dans [BK12], trois grandes catégories importantes de besoins sont identifiées pour répondre aux exigences de l'industrie en matière de scalabilité des outils de MDE :

- La persistance : la capacité d'accès et de mise à jour de très grand modèle avec une utilisation mémoire basse et une rapidité d'exécution correcte.
- Les requêtes et les transformations : la capacité d'exécuter des requêtes et des transformations sur de grands modèles avec un temps d'exécution faible.
- Le travail collaboratif : la capacité pour de multiples développeurs d'accéder, d'exécuter des requêtes, de modifier et de gérer les versions de très grands modèles partagés de manière non invasive.

Dans les sections suivantes, nous nous concentrerons sur la présentation des recherches concernant les deux premières catégories de besoin. En effet, les améliorations concernant le travail collaboratif sortent du cadre de ce travail. Malgré tout, on peut citer deux contributions majeures dans ce domaine que sont ModelBus [SBG08] et EMFStore [KH10]. Toutes deux basées sur l'utilisation de système de versionnement de fichiers (CVS et Subversion) pour l'implémentation de repository pour les modèles.

3.3 Les solutions de persistance

Comme nous l'avons vu, il existe trois grandes catégories d'améliorations importantes pour l'industrie en termes de scalabilité des outils de MDE. Cependant, nous nous concentrerons sur les recherches portant sur les améliorations relatives à la persistance et à l'exécution de requête sur les modèles. Car il s'agit des deux aspects les plus intéressants et les plus transposables pour le sujet de recherche qui nous occupe ici.

Étant donné que la large adoption de l'Eclipse Modeling Framework pour supporter le MDE en a fait un standard de facto et que son abstraction de la couche de persistance permet facilement de changer d'implémentation pour le *model repository*, l'essentiel des recherches dans le domaine de la persistance des modèles a été réalisé dans cet écosystème.

Avant d'aborder les pistes d'amélioration en matière de *model repository* ayant fait l'objet de recherches, il est important de définir ce que représente la persistance d'un modèle dans le monde EMF, et quels sont les facteurs pouvant limiter la scalabilité en matière de persistance et d'exécution de requête dans le monde EMF.

La persistance d'un modèle consiste à transformer un graphe d'objets représentant un modèle en mémoire, en un format adapté à la solution de persistance. À l'inverse, le chargement d'un modèle consiste à reconstruire un graphe d'objets conforme à partir des données provenant de la solution de persistance. Si le graphe d'objets est reconstruit à partir du noeud racine, on parlera d'un *full load*, à contrario si le graphe est reconstruit à partir d'objets arbitraires, on parlera de chargement partiel. [PCM11].

Comme solution de persistance des modèles, EMF utilise par défaut la sérialisation sous forme de fichiers XMI. Même si l'utilisation de ce standard répond de manière adéquate aux besoins d'interopérabilité, il pose certains problèmes en matière de scalabilité. En effet, la sérialisation des modèles dans des fichiers XML présente deux types de problèmes d'efficacité pour les modèles de grandes tailles :

- La verbosité des fichiers qui favorise la lisibilité humaine impacte fortement l'efficacité des inputs/outputs.
- Pour obtenir des modèles navigables, il est nécessaire de parser entièrement les fichiers XMI (effectué via l'api SAX Parser). Il n'est donc pas possible de faire du chargement à la demande. Ce qui impacte fortement l'occupation mémoire nécessaire pour charger les modèles et effectuer des requêtes sur ceux-ci.

3.4 L'approche relationnelle

Les toutes premières solutions pour faire face aux limitations de la sérialisation XMI dans le cadre des modèles de grandes tailles, se sont orientées vers l'utilisation de base de données relationnelles comme solution de persistance. Les deux représentants de cette approche sont CDO et Teneo.

Connected Data Objects (CDO) est la première solution fournissant un mécanisme de chargement à la demande d'éléments de modèles, ainsi que le déchargement automatique de ces éléments lorsque la mémoire devient pleine grâce à l'utilisation de *soft reference* (un objet uniquement référencé par une *soft reference*, peut être supprimé de la mémoire par le garbage collector en réponse à un besoin de libération de mémoire). Cette solution permet d'utiliser la plupart des systèmes de bases de données relationnelles (il est également possible d'utiliser d'autres types de bases de données).

Bien que CDO fournisse une couche d'abstraction entre les applications EMF et le code d'accès aux données spécifiques au système de base de données sous-jacent, son utilisation n'est pas totalement transparente pour les applications. En effet, afin de bénéficier des avantages du *lazy-loading* et du déchargement automatique, il est nécessaire de traiter les métamodèles pour leur adjoindre un certain nombre d'options spécifiques au CDO.

Teneo quant à lui utilise l'ORM Hibernate[7] afin de persister les modèles dans des bases de données relationnelles. Une approche similaire est mise en place dans l'implémentation actuelle dans MetaDone, avec l'utilisation d'OpenJPA[8] (évoquée dans la section 2.3). Teneo fournit également une abstraction du code de la base de données utilisée, en produisant un schéma relationnel ainsi qu'une API EMF à partir d'un métamodèle.

Même si l'approche CDO/Teneo en fournissant une gestion du chargement plus efficace des modèles (chargement partiel et à la demande), représente une avancée par rapport à l'approche XMI, elle reste insuffisante pour gérer de manière optimale les modèles de grandes tailles. En effet, la nature fortement connectée de ce type de modèle nécessite pour les bases de données relationnelles d'exécuter de multiples opérations de jointures pour répondre aux requêtes complexes. Ce qui a pour effet de limiter les performances aussi bien en termes de temps d'exécution que de consommation mémoire[BK12]. Ces problèmes de performance relatifs aux multiples opérations de jointures sont susceptibles d'affecter également le meta-repository de MetaDone. Il est donc pertinent d'explorer les alternatives possibles.

3.5 L'approche NoSQL

3.5.1 Introduction aux technologies NoSQL

Les bases de données dites NoSQL sont nées avec l'émergence des grandes entreprises du web telles que Google et Amazon. Ces entreprises, nécessitant le traitement de très gros volumes de données, ont été confrontées aux limitations en matière d'extensibilité des bases de données classiques. En effet, les bases de données classique n'ont pas été prévues, au départ, pour fonctionner de manière distribuée. Ce qui a comme conséquence que pour répondre à une augmentation du volume de données à traiter, tout en maintenant des performances acceptables, celles-ci doivent se baser sur la scalabilité verticale (ajout de mémoire, utilisation d'un plus gros processeur, etc.).

Pour répondre à leurs besoins de traitement de volumes croissants de données, ces entreprises ont donc mis au point leur propre solution de persistance, telle que BigTable pour Google ou Dynamo pour Amazon, basée sur le modèle de scalabilité horizontale (ajout de serveur). Le respect strict des propriétés ACID représentant un obstacle pour ce type de bases de données distribuées, il a été nécessaire d'introduire pour celles-ci une approche plus souple basée sur l'arbitrage entre la consistance et la disponibilité (Théorème CAP)[SF12].

Il existe quatre grandes catégories de bases de données NoSQL :

- Map : ce type de base de données stocke les données sous forme de paires clé-valeur. Les clés représentent des identifiants uniques, les valeurs sont sans schéma prédéfini, elles peuvent donc être de n'importe quel format texte, HTML, XML, JSON, ... Dans ce type de base de données, seules les recherches par le biais des clés sont possibles.

- Document : Ce type de base de données est assez proche du modèle clé-valeur, à la

[7] `http://hibernate.org/`
[8] `http://openjpa.apache.org/`

différence que la notion de document permet d'effectuer des recherches dans les valeurs et d'établir des relations entre celles-ci.

- Column : Ce type de base de données stocke les données sous forme de lignes (identifiées par une clé unique) qui référencent chacune un ensemble de colonnes. La particularité qui différencie se type de base données du modèle relationnel est que chaque ligne peut avoir un groupement de colonnes différent. Ce qui permet par exemple de gérer les utilisations qui induisent des lignes partiellement vides (*sparsely populated row*).

- Graph : Les données sont représentées sous forme d'un ensemble de noeuds liés ensemble par des arcs (propriété d'*index free adjacency*). Chaque noeud contient des champs de données et les requêtes sur les données du graphe utilisent des algorithmes efficaces de parcours de graphe. Ce modèle de base de données NoSQL est un peu à part, en effet, à contrario des autres types de bases de données NoSQL, ce modèle ne se base pas sur la notion d'agrégat. De plus la naissance de ce type de base de données a été moins influencée par le besoin d'extensibilité que par la nécessité de modéliser et de traiter des données fortement connectées.

3.5.2 L'approche orientée Document

Les bases de données classiques ne permettent donc pas de répondre pleinement au besoin de scalabilité des outils de MDE. Ce qui a conduit les chercheurs à examiner des solutions utilisant l'approche NoSQL. La solution pionnière de cette approche en matière de persistance des modèles fut le prototype Morsa.

Dans l'article présentant Morsa [PCM11], on identifie les caractéristiques avantageuses dont disposent les bases de données NoSQL en comparaison aux bases de données relationnelles. Ces caractéristiques sont :

- Scalabilité : Les bases de données NoSQL se comportent mieux en termes de scalabilité par rapport aux bases de données relationnelles dans le cadre des applications gérant une grande quantité de données représentant des modèles objet. Ce qui est le cas de nombreuses applications MDE gérant des modèles de grande ampleur.

- Schemaless : Contrairement aux bases de données relationnelles, les bases de données NoSql ne nécessitent pas de définition de schéma. Cette caractéristique est particulièrement intéressante, car au sein des repositories relationnels, on crée généralement un schéma pour chaque métamodèle, ce qui rend leur évolution et la conformité des modèles existants aux nouvelles versions de leurs métamodèles, difficiles (ce qui n'est pas le cas de MetaDone).

- Accessibilité : De nombreuses bases de données NoSQL permettent de visualiser leurs données sous forme d'objets JSON grâce à des API qui peuvent être accédées par des appels HTTP. Ce qui permet d'accéder aux modèles via, par exemple, des navigateurs web ou des services web.

Au-delà des caractéristiques générales identifiées du modèle NoSQL, Morsa opte pour l'approche orientée document, en utilisant une base de données MongoDB. Ce type de base de

données permet de mapper de façon simple et naturelle les éléments des modèles à des documents. Il est possible de mapper ces éléments selon différents niveaux de granularité (un modèle par document, un fragment de modèle par document, etc.). Dans Morsa, la granularité choisie pour mapper les modèles à des documents est la plus fine, c'est-à-dire un seul élément du modèle par document. Les attributs d'un élément sont donc persistés sous forme de paires clé-valeur, au côté de ses metadata (tel que les références vers d'autres documents pour lier les éléments entre eux).

Afin d'optimiser le temps d'exécution et la gestion de la mémoire, Morsa utilise un mécanisme de chargement à la demande allié à un cache d'objet qui permet de conserver les objets des modèles en mémoire afin de réduire les requêtes vers la base de données. La gestion de la mémoire est également moins rudimentaire que dans le cas de CDO, car le cache bénéficie de stratégies d'éviction configurables qui permettent de décider quel objet retirer de la mémoire lorsque celle-ci est presque pleine (FIFO, LIFO, LRU, etc).

L'évaluation comparative des performances de Morsa, par rapport aux autres approches, a été faite en exécutant un certains nombres de scénarios. Ces scénarios tentent de reproduire des cas d'utilisation courant dans le monde de MDE. Il s'agit donc d'insertion de modèles de différentes tailles dans le repository, ainsi que des opérations d'accès à ces modèles et d'exécution de requête sur ceux-ci. Les modèles sont ceux qui ont été proposés dans le workshop GraBaTs et conforme au métamodèle Java proposé dans MoDisco [BCJM10]. Ce workshop avait pour objectif de comparer entre autre les performances d'outils de transformation de graphe et de modèles sur un nombre choisi d'études de cas[Gra].

Cette évaluation de Morsa a permis de démontrer qu'une solution NoSQL pouvait se montrer plus adaptée et plus performante pour la gestion des modèles de grandes tailles qu'une approche relationnelle. Cependant, la stratégie de mapping des modèles aux documents induit malgré tout une certaine inefficacité. En effet, cette stratégie a pour inconvénient de devoir implémenter un mécanisme pour lier les objets entre eux avec un genre de clés étrangères. Ceci a pour conséquence d'entraver l'insertion et la vitesse des requêtes, lorsque les éléments des modèles tendent à être fortement interconnectés. Ce qui n'est pas sans rappeler les problèmes de jointures rencontrés dans les approches relationnelles.

EMF fragments [SZFK12] est une des autres solutions utilisant une base de données MongoDB (elle permet également d'utiliser HBase[9] qui est une base de données orientées clé/-valeur). L'approche d'EMF fragments se différencie de celle proposée par Morsa, par la granularité des éléments des modèles à persister. En effet, cette solution partitionne de manière automatique les modèles en fragments, et ce sont ces fragments qui sont persistés ou chargés à la demande. Afin de guider ce processus, il est nécessaire de spécifier au niveau métamodèle des informations de partitionnement. Le désavantage de cette approche est que les performances sont fortement tributaires de la taille des partitions et donc de la configuration spécifiée. Une petite fragmentation donnera des résultats proches de ceux rencontrés avec XMI. À l'inverse, une trop grande fragmentation donnera des résultats similaires à ceux de Morsa.

[9] `http://hbase.apache.org/`

3.5.3 L'approche orientée Graphe

Nous avons vu que toutes les solutions évoquées avaient un point commun. L'inefficacité dans la gestion des modèles possédant des éléments fortement interconnectés. Il est donc tout naturel que ce constat ait conduit à s'intéresser aux bases de données orientées graphe qui de par la nature même de leur modèle de données sont plus adaptées à la gestion des données fortement connexes.

Une première investigation de cette approche fût abordée dans [BK12], avec une étude comparative de deux prototypes. L'un utilisant Neo4J[10] qui est une base de données orientée graphe très populaire, et l'autre utilisant OrientDB[11] qui est quant à elle une sorte de base de données orientée documents dans laquelle les documents et les relations entre eux sont stockés sous forme de graphe.

L'une des caractéristiques particulièrement intéressantes de ce type de base de données, identifiée dans cette étude, est l'*index free adjacency*. Cette caractéristique correspond au fait que chaque noeud possède l'adresse physique de chacun de ses noeuds voisins et donc pour voyager d'un noeud à un autre, il n'est pas nécessaire d'utiliser un index contenant les adresses physiques des noeuds contrairement à ce qui se passe au sein des bases de données relationnelles par exemple. Ce qui peut grandement améliorer les performances lors de l'exécution de requêtes complexes sur les modèles.

Pour questionner ces deux bases de données orientées graphe, deux façons de faire ont été évaluées. L'API native qui implique d'utiliser les indexes définis comme point de départ de la requête et ensuite de naviguer dans les objets du graphe par le biais de leurs relations, pour calculer le résultat requis. L'autre alternative évaluée est l'utilisation d'un langage de plus haut niveau et indépendant de la technologie de persistance, ici en l'occurrence l'*Epsilon Object Language*.

L'évaluation de ces prototypes a été réalisée de la même manière que pour Morsa, en utilisant des benchmarks provenant du workshop GraBaTs. Lors de l'exécution de ces benchmarks, les deux prototypes ont permis de persister des modèles de plus grandes tailles que CDO et Teneo et ont fourni de meilleures performances (temps d'exécution et consommation mémoire) pour l'exécution de requêtes sur les modèles que XMI, CDO et Teneo (aussi bien en utilisant les API natives qu'en utilisant un langage de plus haut niveau).

Une autre solution utilisant la même approche avec une base de données Neo4J est présentée dans [BGS+14] et démontre des résultats similaires.

3.5.4 L'approche orientée Map

Une autre approche intéressante est présentée dans [GTSC15]. Cet article fixe deux objectifs de performance pour une nouvelle piste de solution en matière de persistance des modèles :

- La solution de persistance proposée doit être memory-friendly, en utilisant des mécanismes performants de chargement des éléments à la demande et d'éviction des objets

[10] `http://neo4j.com/`
[11] `http://www.orientdb.org/`

non utilisés.

- La solution de persistance proposée doit surpasser le temps d'exécution des solutions de persistance actuelles lors de l'exécution des requêtes sur des modèles de très grandes tailles en utilisant l'API standard.

Alors que jusque là les solutions proposées se concentraient sur l'utilisation de base de données orientée graphe et documents, la solution proposée dans [GTSC15] se base quant à elle sur l'utilisation d'une base de données orientée map (MapDB[12]) pour répondre à l'exigence de performance définie. Ce type de base de données, de conception et de mise en oeuvre relativement simple fournit des performances intéressantes avec un temps moyen constant pour les opérations de recherche, d'insertion ou de suppression.

Pour cette solution, un soin important a été apporté afin d'implémenter un modèle permettant de stocker, dans une structure aussi rudimentaire qu'un ensemble de clé/valeur, des modèles EMF. La solution se structure donc sous forme de trois maps différentes :

- un property map pour conserver toutes les données des objets de manière centralisée.
- un type map pour conserver la manière dont les objets interagissent entre eux avec le métalevel(instanceof, relationships).
- un containment map, qui définit la structure du modèle en termes de référence containment.

Pour répondre à l'exigence imposée de la gestion de la mémoire efficace, la solution combine deux mécanismes :

- Un mécanisme de lazy-loading semblable à celui mis en place dans les ORM comme Hibernate, en associant à chaque objet, un objet délégué léger (proxy) qui s'occupe du chargement à la demande des données de l'élément et de garder les changements d'état des éléments.
- Pour éviter de conserver des objets des modèles en mémoire non utilisés, la solution utilise, à l'instar de CDO, le mécanisme de *soft reference*, ce qui permet au garbage collector de désallouer n'importe quel objet du modèle qui n'est pas directement référencé par l'application.

Comme les autres solutions jusqu'ici présentées, la solution utilisant une base de données orientée Map a été éprouvée à l'aide de benchmarks provenant de GraBaTs. L'analyse des résultats montre que cette approche se comporte généralement mieux que les autres solutions, même si certains résultats sont tempérés par le mécanisme d'éviction des objets de la mémoire qui entraîne un surcoût en temps d'exécution dû à l'exécution trop régulière du garbage collector.

La présentation faite dans [GTSC15] a également montrée que certains cas d'utilisation pouvaient bénéficier de meilleures performances en utilisant des structures de données, de plus bas niveau que celles fournies par les bases de données relationnelles ou orientées graphe. En effet, les utilisations ne nécessitant pas de calcul de requêtes complexes (essentiellement

[12] `http://www.mapdb.org/`

axées sur l'accès atomique), peuvent tirer parti du temps d'accès constant relativement bas des bases de données orientées map.

3.6 L'approche multiparadigme

Les solutions utilisant une base de données NoSQL, présentées jusqu'ici, ont chacune démontré de bonnes performances dans le cadre de la gestion de la persistance et du chargement des modèles de grandes ampleurs. Cependant aucune n'offre le même niveau de performance sur l'ensemble des activités de modélisation que les outils de MDE.

Ceci s'explique par le fait que chacune de ces approches offre une manière générique de représenter les modèles, adaptée à son type de base de données sans se soucier de la manière d'accéder aux modèles. Par exemple, la représentation de modèle adaptée à une base de données orientées graphe, représente une solution optimale pour rechercher un graphe d'élément dans un modèle, mais est nettement moins adaptée à l'accès atomique d'élément de ce même modèle. Or la manière d'accéder aux modèles est fortement liée à l'activité de modélisation exercée.

Partant, donc, du constat que certains types de bases de données sont plus adaptés que d'autres à une activité de modélisation spécifique et en accord avec la notion de *polyglot persistence* défendue dans [SF12], [Dan16] présente une approche multiparadigme de la persistance des modèles, NeoEMF. Cette approche propose donc l'utilisation de plusieurs types de bases de données, le choix de l'utilisation de l'une ou de l'autre étant motivé par le cas d'utilisation. NeoEMF permet donc d'utiliser, trois types de couches de persistance différents : orientée map, orientée graphe et orientée colonnes.

NeoEMF fournit trois connecteurs, afin d'être capable de représenter les modèles dans chacun de ces types de base de données :

- NeoEMF/Map : Il s'agit du connecteur, évalué dans [GTSC15], conçu pour l'utilisation d'une base de données orientée clé/valeur. Cette couche de persistance a pour objectif de fournir un accès rapide aux opérations atomiques, telles que l'accès à un seul élément/attribut ou la navigation au travers d'une seule référence. Il s'agit de la solution la plus appropriée pour améliorer les performances et la scalabilité des outils qui ont besoin d'accéder à de très grands modèles sur une seule machine.

- NeoEMF/Graph : Ce connecteur a été repris des travaux menés sur Neo4EMF exposés dans [BGS+14]. Il est donc conçu pour l'utilisation d'une base de données orientée graphe. Cette approche est indiquée dans les cas d'utilisation nécessitant le calcul efficace de requêtes complexes sur des modèles (traduit en parcours de graphe).

- NeoEMF/Column : Ce connecteur est conçu pour l'utilisation d'une base de données distribuée orientée colonnes. Cette solution offre des accès concurrents en lecture/écriture et garantit les propriétés ACID au niveau des éléments des modèles (agrégats). Cette approche a pour but de permettre le développement d'application MDE distribuée.

Pour répondre au mieux au besoin de performance et de scalabilité, NeoEMF utilise des mécanismes de chargement à la demande et des stratégies de caching configurables. NeoEMF étend également le support des caches en intégrant un framework de prefetching/caching appelé PrefetchML. Ce framework fournit une DSL permettant de définir des règles spécifiques de caching et de préchargement [DSC16].

En bref, le framework NeoEMF avec son écosystème riche, représente actuellement l'état de l'art en matière de repository pour les applications EMF.

3.7 Perspectives pour l'amélioration des performances du metarepository

Un certain nombre de travaux portant sur une problématique proche de celle au coeur de ce travail ont été présentés dans la section précédente. Dans la suite, il sera détaillé les possibilités d'amélioration des performances du metarepositoy. Ces pistes de solution sont issues de la réflexion menée sur base de l'étude des travaux précédents, et sur des observations menées sur l'implémentation actuelle du metarepository.

3.7.1 Changement de paradigme

Bon nombre de travaux menés sur EMF, conduisent à un changement de paradigme en matière de modèle de données de persistance. En effet, la nature fortement connectée des modèles remet en cause l'utilisation du modèle relationnel, car elle implique d'effectuer de nombreuses opérations de jointures (opérations qui peuvent se révéler coûteuses). Cette caractéristique se retrouve également dans le metarepository de MetaDone (impliquant des relations reflexives sur l'élément *DataObject*). Dès lors, adopter une approche similaire semble pertinent.

Les travaux paraissant les plus prometteurs dans cette optique, sont ceux portant sur l'approche orientée Graphe et sur l'approche orientée Map. Pour répondre aux problèmes des opérations de jointures, l'approche utilisant une base de données graphe tire sa force de son formalisme particulièrement adapté pour représenter des données fortement connectées et sur le concept *index free adjacency*. L'approche orientée map quant à elle se fonde sur l'utilisation d'une structure de données rudimentaire et performante, mais nécessite un travail plus important de conception de la couche de représentation en base de données (en l'occurrence pour EMF la représentation est conçue sous forme de trois maps différentes).

3.7.2 Mécanisme de caching et de gestion mémoire

Il a également été constaté dans les travaux portant sur les problèmes de scalabilité dans l'écosystème EMF, qu'une optimisation de la gestion de la mémoire et d'utilisation de mécanisme de cache pouvaient influencer positivement les performances de la couche de persistance. En effet, ces mécanismes permettent d'éviter les allers-retours entre l'application et la base de données (caching), et d'éviter de surcharger la mémoire en y conservant des objets devenus inutiles pour la tâche en cours.

Cependant il est à noter que bon nombre de ces mécanismes sont déjà mis en oeuvre par l'ORM utilisé dans MetaDone. En effet, des mécanismes de chargement à la demande sont déjà présents dans OpenJPA (*lazy-loading*), et en matière de caching le contexte de persistance géré par l'*EntityManager* représente un cache de premier niveau qui permet de limiter les accès à la base de données.

Il est toutefois possible d'adjoindre un deuxième niveau de cache à ce premier niveau via une implémentation d'un *cache provider*, tel que EHCache[13] ou JBossCache[14].

3.7.3 Changement d'implémentation d'ORM

Comme expliqué dans la section 2.3, l'implémentation actuelle se base sur un ORM implémentant l'API JPA. Il est à noter que chaque implémentation du standard JPA possède ses particularités. Ainsi un framework peut, par exemple, choisir d'implémenter des extensions à la spécification ou de n'implémenter qu'un sous ensemble de celle-ci. Cette variabilité peut également affecter d'autres éléments moins visibles mais tout aussi importants tel que la performance.

L'implémentation OpenJPA utilisée dans MetaDone ne semble pas celle offrant les meilleures performances par rapport à d'autres implémentations disponibles sur le marché si l'on en croit par exemple [Rub10] où les résultats de benchmarks établis par un concurrent [Sof12]. Il pourrait donc être judicieux de changer d'implémentation JPA pour le metarepository.

3.7.4 Résoudre les problèmes d'implémentation de l'API

Il existe également un certain nombre de problèmes directement liés à la manière d'implémenter l'API du metarepository. Parmi ceux-ci, on peut citer :

- La gestion des listes : les listes sont retournées par copie au code appelant de l'API. Cette solution représente une bonne pratique en terme d'encapsulation. Cependant, cette manière de faire s'avère coûteuse dans le cas de listes de grande taille.

- La vérification de l'unicité des noms des *DataObjects* : L'implémentation de la vérification de la contrainte d'unicité des noms de *DataObject* se fait de manière programmatique en parcourant tous les DataObjects. Il serait plus performant de laisser gérer ceci par la base de données via une contrainte d'unicité, et de traduire l'exception lancée par la base de données en une exception business (*BadPreCondition*).

Afin d'identifier et résoudre ce genre de problèmes dans l'implémentation du metarepository, il serait intéressant d'appliquer une méthodologie d'optimisation de performance de code basée sur la construction de tests de performance ainsi que l'utilisation d'outils de profilage de code.

[13] `http://www.ehcache.org/`
[14] `http://jbosscache.jboss.org/`

3.7.5 Construire MetaDone au-dessus du framework EMF

Une des solutions possibles est également de repenser MetaDone pour le construire au-dessus du framework EMF. Ce qui permettrait de bénéficier des optimisations issues des nombreuses recherches menées sur cet environnement, et de l'écosystème important d'EMF. Il s'agit donc d'une refonte assez radicale et un travail très ambitieux qui sort un peu du cadre de ce travail. Cependant, cette solution pourrait être une solution à envisager dans un travail futur.

3.7.6 Option choisie

Il a été présenté un certain nombre de possibilités qui serait susceptible d'améliorer les performance du metarepository par rapport à la solution actuelle. Cependant certaines d'entre-elles, semblent moins prometteuses que d'autres.

La marge d'amélioration possible offerte par un système de caching et gestion mémoire optimisée semble faible, car bon nombre de mécanismes sont déjà offerts par l'implémentation actuelle.

Le changement d'implémentation de l'ORM, pourrait représenter une solution intéressante. Cependant, les travaux menés sur Teneo et présentés dans la section 3.4 montre que conserver une approche relationnelle pour la base de données, pourrait ne pas être la meilleure option pour répondre au besoin de scalabilité de MetaDone. Cette raison motive également le choix d'écarter, pour l'instant, les optimisations à mener sur l'implémentation de l'API, même si cela représente une solution pouvant s'avérer gagnante.

L'option qui sera présentée dans la suite, est donc celle d'un changement de paradigme. Cependant, elle n'implique pas l'utilisation d'un des types de base de données mentionnés. Elle se base sur une autre alternative qui permet d'apporter une contribution intéressante.

Nous verrons que cette alternative est une approche prometteuse en terme de performance, qui tire parti de caractéristiques proches de celles du modèle orienté graphe (présentés dans la section 3.5.3). Celle-ci permettra également une réutilisation de nombreux éléments qui avaient déjà été développés dans l'implémentation actuelle, en cela, elle est proche de la piste de solution consistant à changer d'implémentation de l'ORM.

4 Alternative au modèle relationnel pour l'implémentation du metarepository

4.1 Introduction aux bases de données orientées objet

Au milieu des années 1980, le modèle relationnel connaît un très grand succès et s'avère très bien adapté aux applications traditionnelles (de gestion). Cependant l'apparition de nouveaux types d'application plus complexes tels que les outils CAD (*computer aided design*), CASE (*computer aided software engineering*), GIS (*geographic information system*), ou encore d'autres outils de gestion des données multimedia, font naître de nouveaux besoins. Parmi ceux-ci, l'on peut citer le besoin de structures de données plus complexes, le besoin de transactions de plus longue durée, ainsi que de nouveaux types de données. Nouveaux besoins pour lesquels les bases de données relationnelles de l'époque semblent moins appropriées [Ban88].

À la même époque, naît également la question de l'*impedance mismatch* entre le langage relationnel et les langages de programmation. C'est-à-dire le décalage entre d'une part, le monde relationnel structuré pour les requêtes ad hoc, et d'autre part, le monde des langages de programmation utilisés dans le cadre du développement d'application de gestion. Mondes possédant chacun leurs types de données, leurs modèles de calcul, etc. Ceci rend complexe l'implémentation d'applications nécessitant l'accès à une base de données, car celles-ci obligent de jongler avec deux types de structures de données. De plus, ce décalage peut également avoir un impact sur les performances car les modèles de calcul différents, parfois mal compris par les développeurs, peuvent conduire à des implémentations inefficaces [Ban88].

Un certain nombre de recherches tentent alors d'apporter une solution à ces nouveaux besoins et de réduire le décalage entre le modèle de persistance de données et le formalisme du langage de programmation utilisé. Ces recherches s'orientent dès lors vers le paradigme orienté objet doté de caractéristiques intéressantes par rapport au modèle relationnel. Parmi celles-ci, l'on peut citer trois caractéristiques importantes :

- Expressivité : Le modèle relationnel impose de représenter les données sous formes de tuples et de relations, ce qui rend la modélisation de données hiérarchiques ou composées ardue. De plus, la représentation de ce type de données complexes réparties sur plusieurs relations, implique la nécessité d'utiliser des opérations de jointures afin de récupérer les données, opérations qui peuvent se révéler particulièrement coûteuses. A contrario, l'approche OO offre une plus grande expressivité grâce à ces concepts d'héritage, de composition et d'encapsulation. Les structures complexes peuvent donc être plus aisément représentées et ne doivent pas être mappées sur une structure de bas niveau.[Ban88] (Avantage qui peut-être mis en parallèle avec la notion d'aggrégat développée dans le cadre de l'approche dite NoSql dans les bases de données orientées documents évoquée dans la section 3.5.1.)

- Extensibilité : Les bases de données relationnelles offrent un ensemble restreint de types de données et ne permet pas, à l'époque, de définir de nouveaux types de données. L'ap-

proche OO avec ses concepts (classe, d'héritage, polymorphisme, ...), fournit, quant à elle, la possibilité de créer de nouveaux types de données à partir de types existants. Cette possibilité d'ajouter de nouveau type/classe permet dès lors d'étendre les capacités d'un système. [Ban88]

- Possibilité de faire disparaître l'*impedance mismatch* relationnel/OO : En effet, le passage vers une représentation OO pour la persistance des données permettrait d'unifier les deux types de modèles de données (persistance et applicative).

Les recherches inspirées des avantages du paradigme OO, ont abouti au concept de bases de données orientées objet. Dans ce type de système, les données sont donc structurées sous forme d'objets plutôt que de tables. La figure 5 provenant de [Alz16], illustre la différence de représentation d'une même réalité entre l'approche OO et l'approche relationnelle.

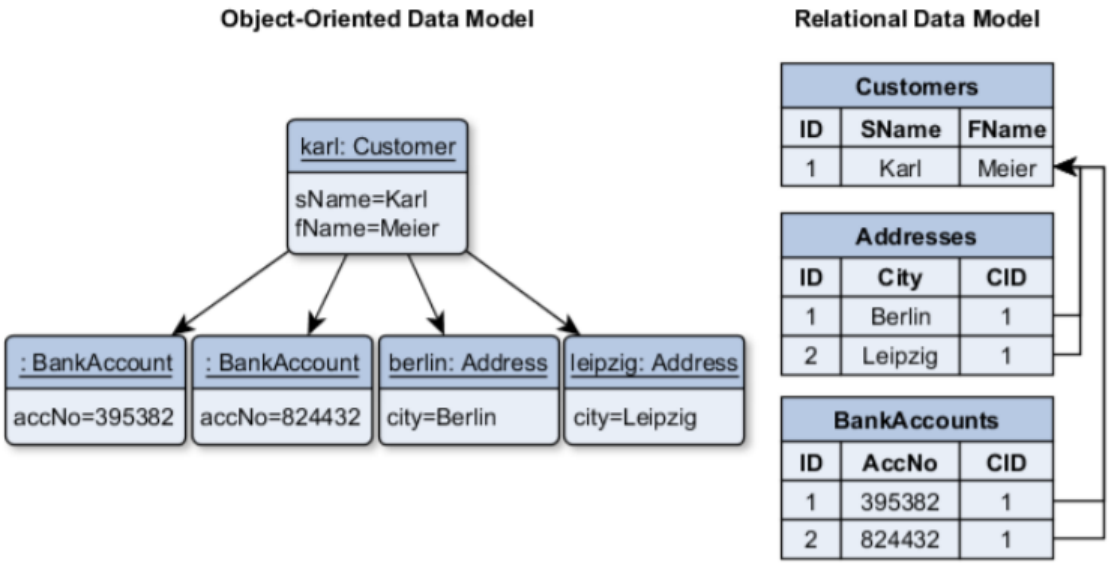

FIGURE 5 – Exemple de représentation de données Relationnelles Vs. Objet provenant de [Alz16]

En dépit d'un certain nombre d'avantages, aussi bien au niveau des facilités de modélisation offertes qu'au niveau d'un gain de performance dans certains cas (point sur lequel nous reviendrons), les bases de données orientées objet n'ont pas connu une aussi large adoption que les bases de données relationnelles. Si certaines des causes de ce manque de succès sont inhérentes au fondement de ces bases de données (manque de fondations théoriques aussi solides que celles du monde relationnel et manque de standardisation), il s'explique essentiellement par le manque de maturité de ces systèmes par rapport à celle des systèmes relationnels. De fait, alors que de nombreuses bases de données OO ne sont finalement que des systèmes de stockages persistants pour un langage de programmation orienté objet spécifique (ce qui est le cas de l'implémentation utilisée dans la suite de ce travail), les bases de données relationnelles, elles, fournissent un vaste ecosystème pour la gestion des données au sens large du terme (comme l'illustre l'ensemble des langages définis DML, DDL, DQL,DCL).

La tendance, par la suite, a été plutôt d'intégrer les éléments positifs des bases de données orientées objet, dans le monde relationnel avec une approche plus hybride qu'un changement

radical comme l'atteste le concept de bases de données orientées relationnelles-objet ou les ajouts contenus dans le standard SQL-99.

4.2 Interêt dans le cadre du metarepository de MetaDone

L'aspect le plus intéressant pour l'implémentation du metarepository de MetaDone, est celui de la performance, car les aspects concernant les facilités de modélisation et suppression de l'*impedance-mismatch* conceptuel sont déjà résolus par l'implémentation actuelle à l'aide d'un OpenJPA (ORM présenté dans la section 2.3).

Effectivement, la structuration des données dans les bases de données orientées objet peut permettre d'obtenir des accès plus performants aux données. Ceci s'explique de part le fait que la récupération de données réparties sur plusieurs relations consiste à suivre des pointeurs entre les différents objets et non plus à effectuer des jointures qui peuvent être des opérations coûteuses. De manière plus détaillée, [Kim93] identifie deux mécanismes en oeuvre dans ce type de bases de données qui représente un avantage en terme de performance par rapport au système relationnel :

Pour illustrer le premier mécanisme, prenons l'exemple d'une instance d'une classe A qui possède un attribut de classe B. Lorsque l'on charge en mémoire cette instance, celle-ci possède une référence vers une instance de la classe B. Par référence, on entend un identifiant de l'objet qui représente soit une adresse physique en mémoire soit une adresse logique. Si l'application souhaite ensuite récupérer l'objet de classe B, et que l'identifiant correspond à une adresse physique alors l'objet peut être directement extrait de la base de données, si par contre il s'agit d'une adresse logique, l'objet doit être récupéré en recherchant une entrée dans un genre de table de hachage.

A contrario dans les bases de données relationnelles, si un tuple x d'une relation A possède un attribut qui fait référence à un tuple d'une relation B, lorsque l'application à récupéré le tuple de la relation A, et qu'elle souhaite ensuite récupéré le tuple de la relation B référencé, il est nécessaire d'exécuter une nouvelle requête. Dans le cas où il n'existe pas d'index sur l'attribut alors l'exécution de la requête nécessite le parcours séquentiel de la relation pour trouver le tuple correspondant. Par contre si un index est maintenu (ce qui est généralement le cas des clés étrangères explicites pour les bases de données relationnelles modernes) alors cela ramène à une recherche dans celui-ci (table de hachage, BTree, ...). Ceci démontre que l'avantage pour les bases de données orientées objet existera surtout si elle possède une gestion par adresse physique (avantage similaire au concept d'*index free adjacency* des bases de données orientées graphes présenté dans la section 3.5.3) évitant ainsi la recherche dans une table de hachage.

Le deuxième mécanisme s'explique de part le fait que la plupart des bases de données orientées objet convertissent les identifiants d'objets stockés dans un objet en pointeurs de mémoire lorsque l'objet est chargé depuis la base de données. Supposons donc que les deux objets X et Y ont été chargés en mémoire, et que l'identifiant de l'objet stocké comme valeur de l'attribut A de l'objet X est converti en pointeur de mémoire virtuelle vers l'objet Y en mémoire. Alors, l'accès de l'objet X à l'objet Y, via son attribute A, revient à effectuer un

simple accès via un pointeur en mémoire. Dans le cas où l'on transpose cela à des centaines voir des milliers d'objets chargés en mémoire, et que chaque objet contient des pointeurs vers un ou plusieurs autres objets en mémoire, on peut facilement imaginer que ce mode de fonctionnement permet d'améliorer les perf.ormances des applications qui nécessitent la navigation répétée à travers les objets liés chargés en mémoire.

Ces mécanismes qui permettent d'éliminer la plupart des opérations de jointure, pourraient être une solution pour l'implémentation du metarepository, puisque la structure de ce repository est caractérisée (comme illustré dans la section 2.3) par un petit nombre de concepts mais possèdant de nombreuses relations entre-eux dont des relations recursives. D'autres types de bases de données, comme les bases de données orientées graphes, permettraient probablement d'obtenir également des gains de performance en jouant sur le même élément. Cependant l'approche orientée objet a pour avantage de ne pas nécessiter de changement radical du design du repository.

4.3 Présentation d'ObjectDB

Maintenant que nous avons vu les éléments qui ont orienté le choix du paradigme OO pour l'implémentation du metarepository de MetaDone, voyons l'implémentation choisie.

ObjectDB[15] est une base de données orientée objet entièrement implémentée en Java. Pour être plus juste et conformément à ce qui a été présenté dans la section 4.1, il faudrait plutôt parler d'un système de stockage persistant destiné aux applications Java. En accord avec cette définition, ObjectDB ne fournit pas d'API propriétaire spécifique. Pour l'utiliser, il est donc nécessaire de passer soit par l'API JPA soit par l'API JDO (*Java Data Objects* qui est un autre grand standard d'accès aux sources de données dans le monde Java[16]).

L'utilisation de l'API JPA recommandée pour l'accès aux bases de données ObjectDB, représente un avantage pour le changement d'implémentation du metarepository de MetaDone. En effet, cela permet une réutilisation importante de ce qui a déjà été implémenté pour la solution actuelle de persistance (OpenJPA/H2), et qui permettra donc de réduire les coûts induits par le changement d'implémentation.

ObjectDB permet deux modes de fonctionnement : le mode embedded et le mode client-serveur. Le fait que le même mode de fonctionnement que l'implémentation actuel soit possible est également un atout, que ce soit en terme de mise en place ou en terme de comparaison des performances (effectivement, l'utilisation en mode client-serveur pourrait avoir un impact sur les performances dû à un overhead induit par le protocole de communication).

Il existe cependant une petite limitation concernant la licence d'utilisation d'ObjectDB pour remplacer la base de données H2. Tandis que H2 est une base de données open source (sous licence Eclipse et Mozilla)[17], ObjectDB n'est clairement pas open source et possède plusieurs niveaux de licence[18]. En ce qui concerne la version testée dans ce travail, il s'agit de la

[15] http ://www.objectdb.com/
[16] http ://www.oracle.com/java/technologies/java-data-objects.html
[17] http ://www.h2database.com/html/license.html
[18] http ://www.objectdb.com/database/purchase

version gratuite, libre d'utilisation pour tous types de projets y compris commerciaux, avec les restrictions suivantes : l'application ne doit ni excéder 10 classes d'entité, ni un million d'objets par base de données.

4.4 Intégration d'ObjectDB à MetaDone

4.4.1 Développement d'un Proof of Concept

Avant de se lancer dans l'intégration d'ObjectDB dans MetaDone, il était important d'appréhender le fonctionnement et la configuration d'ObjectDB, ce qui a permis de facilité la suite du travail et d'également vérifier la viabilité de cette option. Il m'a donc paru intéressant de réaliser un Proof of Concept basé sur un domaine plus rudimentaire afin de s'abstraire de la complexité inhérente au domaine de MetaDone[19].

Ce projet avait non seulement pour objectif d'acquérir de l'expérience sur l'outil durant la phase exploratoire, mais également de permettre de tester certaines hypothèses concernant les solutions à apporter aux problèmes éventuels survenant lors de l'intégration dans MetaDone.

4.4.2 Implémentation dans MetaDone

Une fois la phase exploratoire réalisée, l'implémentation du nouveau metarepository a pu commencer. Dans l'itération suivante, il a surtout été question de reprendre le travail du professeur Vincent Englebert contenant les premières briques de la nouvelle implémentation (bundle metadone_bundle_odb) et d'ajouter la configuration nécessaire dans le client pour pouvoir démarrer l'application en utilisant la nouvelle implémentation. Cette itération a permis de mettre à jour un certain nombre de problèmes concernant les différences d'implémentation de la spécification JPA (mapping ou autres).

4.4.3 Résolution des problèmes liés au changement d'implémentation JPA

En premier lieu, il est à mentionner que les deux implémentations utilisent deux versions différentes du standard JPA. En effet, alors que OpenJPA 2.3.0 implémente la spécification dans sa version 2.0 la version d'ObjectDB choisie (version 2.8.2) quant à elle implémente la version 2.1. Néanmoins, cette modification de version du standard n'a pas eu réellement d'impact pour notre intégration. Par contre nous verrons dans cette section que chaque implémentation a pris parfois des libertés par rapport à la spécification, soit en permettant l'utilisation de constructions non permises par le standard, soit en ne l'implémentant que partiellement.

[19] Accessible sur http ://github.com/Masterjim/objectdb-poc

4.4.3.1 La déclaration d'index

Le premier problème de mapping rencontré concerne la déclaration d'index. Dans les faits, le code de l'implémentation OpenJPA défini des index sur certains attributs pour rendre la recherche plus efficace. A cette fin, OpenJPA fournit une annotation propriétaire, l'annotation @Index[20]. Celle-ci permet donc de définir la construction d'un index pour l'attribut annoté.

Bien évidemment ObjectDB, n'utilise pas cette annotation. Cependant la version 2.1 de la spécification JPA défini une nouvelle annotation pour répondre au même besoin[21]. Seule sa mise en oeuvre diffère quelque peu de la manière de celle d'OpenJPA, comme nous pouvons le constater dans l'exemple ci-dessous

```
@Entity
public class DataObjectJPA implements DataObject {
    ...

    @Basic(optional = true)
    @Index(name = "name_index")
    @Column(length = 100)
    private String name;

    ...
}
```

Listing 1 – Implémentation d'un index avec OpenJPA

```
@Entity
@Table(indexes = {@Index(name="name_index", columnList = "name")})
public class DataObjectODB implements DataObject {
    ...

    @Basic(optional = true)
    @Column(length = 100)
    private String name;

    ...
}
```

Listing 2 – Implémentation d'un index en suivant le standard JPA

Malheureusement, ObjectDB ignore purement et simplement l'annotation JPA (donc sans message d'avertissement). La documentation d'ObjectDB recommande d'utiliser l'annotation correspondante JDO pour les constructions d'index et ce même si nous nous trouvons dans un contexte JPA[22].

Fait intéressant, la documentation donne quelques informations concernant l'implémentation des index. En effet, les index sont gérés à l'aide d'un BTree maintenu sur le système de fichier

[20] http://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/ref_guide_mapping_jpa.html#ref_guide_mapping_jpa_index

[21] Accessible sur le site d'Oracle http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html

[22] http://www.objectdb.com/java/jpa/entity/index

et non en mémoire. Les clés des BTree sont toutes les valeurs uniques de l'attribut indexé et que chaque clé est associée à une liste de référence vers les entités qui contiennent cette valeur. La documentation donne également quelques bonnes pratiques concernant l'utilisation des champs indexés dans les requêtes (ordre des arguments dans la clause where par exemple). Ces recommandations sont assez similaires aux recommandations dans l'utilisation d'index dans le monde relationnel comme celles que l'on peut trouver dans [Win12].

4.4.3.2 La persistance des *FacetType*

La deuxième construction propriétaire qu'il a fallu transposer dans l'implémentation utilisant ObjectDB est une construction OpenJPA qui permet de personnaliser la manière dont un type ou un objet doit être persité en bases de données et chargé depuis la base de données (serialisation/désérialisation).

```
@Entity
public class DataObjectJPA2 implements DataObject {
    ...
    @Persistent
    @Externalizer("jpaExtern")
    @Factory("jpaBuild")
    private Facets DataObject_facets;
    ...
}

public class Facets {
    private final EnumSet<FacetType> facets;
    ...
    public static int jpaExtern(Facets fs) {
        EnumSet<FacetType> types = fs.getFacets();
        int result = 0;
        for (FacetType t : types) {
            result |= 1 << t.ordinal();
        }
        return result;
    }

    public static Facets jpaBuild(int types) {
        Facets fs = new Facets();
        EnumSet<FacetType> result = fs.getFacets();
        FacetType[] values = FacetType.values();
        for (byte i = 0; types != 0; i++, types >>= 1) {
            if ((types & 1) != 0) {
                FacetType t = values[i];
                result.add(t);
            }
        }
        return fs;
    }
}
```

Listing 3 – Personalisation du marshalling/unmarshalling des *FacetType* dans MetaDone

Cette configuration permet de gérer plus efficacement la persistance des facets d'un DataObject MetaDone. En effet, au lieu de persister cet ensemble sous forme d'une relation "un à plusieurs" entre d'une part la table DataObjectJPA et d'autres part une table Facets, l'ensemble des facets sont stockés dans la table DataObjectJPA sous forme d'un entier. L'accès est également rendu plus performant car il ne nécessite pas de jointures.

Les annotations provenant d'OpenJPA ne faisant pas partie du standard JPA, elles ne sont pas présentes dans ObjectDB. ObjectDB ne fournit pas non plus d'implémentation spécifique facilitant ce type de construction. Etant donné que le problème était plus complexe que le point précédent, il semblait judicieux de reproduire ce mécanisme dans le domaine simplifié du POC mis en place dans la première phase d'intégration afin de facilité l'exploration de possibles solutions.

Cette experimentation à d'abord concerné un des mécanismes introduit dans la spécification 2.1 de JPA. En effet, JPA à partir de cette version fournit un mécanisme élégant pour répondre au même besoin. La création d'une classe se chargeant de la conversion et implémentant l'interface *AttributeConverter*[23]. Pour que cette construction soit reconnue par l'implémentation, il est nécessaire de lui adjoindre l'annotation *Converter*[24], et enfin référencer cette construction dans l'entité. Le code correspondant se trouve ci-dessous :

```
@Converter
public class FacetsAttributeConverter implements
   AttributeConverter<Facets, Integer> {

    @Override
    public Integer convertToDatabaseColumn(Facets attribute) {
        int result = 0;

        EnumSet<FacetType> types = attribute.getFacets();
        for (FacetType t : types) {
            result |= 1 << t.ordinal();
        }

        return result;
    }

    @Override
    public Facets convertToEntityAttribute(Integer dbData) {
        int types = dbData;

        Facets fs = new Facets();
        EnumSet<FacetType> result = fs.getFacets();
        FacetType[] values = FacetType.values();
        for (byte i = 0; types != 0; i++, types >>= 1) {
            if ((types & 1) != 0) {
                FacetType t = values[i];
                result.add(t);
            }
        }
```

[23] http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeConverter.html
[24] https://docs.oracle.com/javaee/7/api/javax/persistence/Converter.html

```
        return fs;
    }
}

@Entity(name="DataObject")
@Table(indexes = {@Index(name="index_name", columnList = "name", unique =
    false)})
public class DataObjectODB implements DataObject {

    ...

    @Column
    @Convert(converter = FacetsAttributeConverter.class)
    private Facets DataObject_facets;

    ...

}
```

Listing 4 – Exemple d'implémentation de la configuration pour les *FacetType* conforme à JPA

Cependant, l'implémentation de ce mécanisme dans le POC a permis de découvrir que cette construction ne semble pas implémentée par ObjectDB. Ceci est confirmé par la consultation du forum d'ObjectDB, où il est stipulé que cet aspect de la spécification 2.1 n'est pas implémenté et ne paraît pas à l'ordre du jour (laissé en suspens depuis plus de 6 ans)[25]. La construction recommandée par l'équipe de support d'ObjectDB est de désactiver la gestion de la persistance avec l'annotation @Transient sur la représentation de l'application, d'ajouter un champs persisté correpondant à la représentation entière et enfin d'utiliser les méthodes callback sur le cycle de vie des entités JPA (pre-persist et post-load) pour assurer la concordance entre les deux avant la persistance et juste après le chargement depuis la base de données. Le code ci-dessous illustre cette construction dans le cadre de MetaDone.

```
@Entity(name="DataObject")
public class DataObjectODB implements DataObject {

    ...

    @Transient
    private Facets DataObject_facets;

    private int DataObject_facets_DatabaseRepresentation;

    ...

    @PrePersist
    public void prePersist() {
        EnumSet<FacetType> types = DataObject_facets.getFacets();
        int value = 0;
        for (FacetType t : types) {
            value |= 1 << t.ordinal();
```

[25] http://www.objectdb.com/forum/842

```
        }

        DataObject_facets_DatabaseRepresentation = value;
    }

    @PostLoad
    public void postLoad() {
        Facets fs = new Facets();
        EnumSet<FacetType> result = fs.getFacets();
        FacetType[] values = FacetType.values();
        int types = DataObject_facets_DatabaseRepresentation;
        for (byte i = 0; types != 0; i++, types >>= 1) {
            if ((types & 1) != 0) {
                FacetType t = values[i];
                result.add(t);
            }
        }
        DataObject_facets = fs;
    }

    ...
}
```

Listing 5 – Exploitation des entity lifecycle events pour la persistance des *FacetType*

L'inconvénient de cette solution est qu'elle force la classe *DataObjectODB* à manipuler la représentation contenue dans la classe *Facets* ce qui viole le principe d'encapsulation. De plus même si cette construction semble fonctionner de prime abord, la multiplication des tests (en ajoutant des éléments, en supprimant des éléments, etc.), révèle que les deux représentations à certains moments ne concordent plus. Ce phénomène s'explique par le fait qu'une fois qu'une instance de *DataObjectODB* est persistée, aucune modification de Facets n'entraîne de modification de la valeur entière correspondante à la valeur en base de données. Pour résoudre ce problème, il faut donc que chaque méthode de modification de contenu de la classe *Facets*, déclenche le recalcul de la valeur entière.

Pour permettre de résoudre le problème de violation du principe d'encapsulation et assurer que les valeurs en base de données et dans l'application concordent, il a été nécessaire d'exploiter les annotations @Embeddable[26] et @Embedded[27] de la spécification JPA. Ces annotations permettent de spécifier que le contenu d'une classe est destiné à être embarqué dans une autre entité. Le code ci-dessous illustre la solution finale pour ce problème.

```
@Embeddable
public class Facets {

    @Transient
    private final EnumSet<FacetType> facets;

    @Basic
    private int dataObjectFacets;
```

[26] http://docs.oracle.com/javaee/7/api/javax/persistence/Embeddable.html
[27] http://docs.oracle.com/javaee/7/api/javax/persistence/Embedded.html

```
    ...

    public void addFacet(FacetType facetType) {
        facets.add(facetType);
        dataObjectFacets = toInt(facets);
    }

    public void addAllFacets(Collection<FacetType> facetTypes) {
        facets.addAll(facetTypes);
        dataObjectFacets = toInt(facets);
    }

    public void removeAllFacets(Collection<FacetType> facetTypes) {
        facets.removeAll(facetTypes);
        dataObjectFacets = toInt(facets);
    }

    static int toInt(EnumSet<FacetType> facets) {
        int result = 0;

        for (FacetType t : facets) {
            result |= 1 << t.ordinal();
        }

        return result;
    }

    static EnumSet<FacetType> toFacets(int types) {
        EnumSet<FacetType> result = EnumSet.noneOf(FacetType.class);

        FacetType[] values = FacetType.values();
        for (byte i = 0; types != 0; i++, types >>= 1) {
            if ((types & 1) != 0) {
                FacetType t = values[i];
                result.add(t);
            }
        }

        return result;
    }

}

@Entity(name="DataObject")
@Table(indexes = {@Index(name="index_name", columnList = "name", unique =
    false)})
public class DataObjectODB implements DataObject {

    ...

    @Embedded
    private Facets DataObject_facets;

    ...
```

```
    @PostLoad
    public void onPostLoad() {
        DataObject_facets.loadFacets();
    }
```

Listing 6 – Solution pour la persistance des FacetTypes

L'implémentation du callback pour l'événement PrePersist n'est plus nécessaire car tout ajout ou suppression de FacetType est directement repercuté sur la valeur entière et donc sa sauvegarde en base de données. Par contre pour assurer qu'au chargement la représentation dans l'application soit cohérente avec celle provenant de la base de données, il est nécessaire de conserver l'implémentation concernant l'événement *PostLoad*. Il est nécéssaire de conserver l'implémentation de cette méthode dans la classe *DataObjectODB* car la gestion des événements du cycle de vie n'est pas possible pour les éléments annotés par @Embeddable dans ObjectDB.

Il serait intéressant dans l'évaluation des performances de vérifier que cette construction apporte un réel gain de performance pour l'implémentation ObjectDB.

4.4.3.3 Type des collections supportées

L'ancienne implémentation utilisait comme type déclaré pour certaines collections (mapping One-To-Many), le type concret *LinkedList*. L'utilisation du type concret *LinkedList* était justifiée dans la classe *DataObjectJPA2* par l'utilisation de méthodes spécifiques de son interface (*addFirst* et *addLast*).

OpenJPA autorise cette utilisation de type concret pour la déclaration de variables destinées au mapping des relations un à plusieurs, cependant il s'agit d'une liberté prise par rapport à la spécification JPA. En effet, la spécification recommande d'utiliser les interfaces comme type déclaré pour les collections (*List* pour les listes, ou *Set* pour les ensembles).

L'implémentation ObjectDB conformément à la spécification n'autorise pas l'utilisation des types concrets. Il a donc été nécessaire de remplacer les types déclarés pour certaines collections. Comme évoqué, l'utilisation du type concret *LinkedList* était justifié dans MetaDone pour bénéficier de méthodes spécifiques. Pour reproduire le comportement dans l'implémentation ObjectDB, il a donc été nécessaire d'implémenter ces méthodes en utilisant uniquement les méthodes déclarées dans l'interface *List*.

4.4.3.4 Mapping des collections d'énumération

L'implémentation utilisant OpenJPA utilisait également une annotation propriétaire pour mapper une collection d'énumération représentant certains aspects comportementaux de propriétés appartenant à un *DataObject*.

```
@Entity
public class DataObjectJPA2 implements DataObject {
```

```
    ...

    @PersistentCollection
    private Set<Kind> PropertyType_kind;

    ...
}

public enum Kind {
    /**
     * if we delete a domain, then we delete the range (and the property)
     */
    A_B,
    /**
     * if we delete the range, then we delete the domain (and the
        property)
     */
    B_A,
    /**
     * if we delete the property, then we delete the domain
     */
    p_A,
    /**
     * if we delete the property, then we delete the range
     */
    p_B;

}
```

Listing 7 – Utilisation de l'annotation @PersistentCollection dans l'implémentation de DataObject

Cette annotation correspond au même comportement que l'annotation @OneToMany, sauf que les entités contenues dans la collection n'ont pas besoin d'être définies comme des entités JPA. Les valeurs énumérées sont alors représentées en base de données par leur valeur ordinale.

De son côté ObjectDB, ne reconnaît pas cette annotation. Cependant aucune annotation n'est à utiliser pour ce cas précis, car le modèle orienté objet d'ObjectDB permet de persister la collection telle qu'elle est représentée dans l'application. Cela a pour avantage d'éviter l'utilisation d'une jointure pour le chargement depuis la base de données.

4.4.3.5 Utilisation de l'API *CriteriaBuilder*

Le dernier problème relatif à la spécification JPA rencontré est l'utilisation de l'API *CriteriaBuilder*[28] pour la construction de certaines requêtes JPA. En effet l'implémentation dans OpenJPA semble plus souple qu'ObjectDB, en ne nécessitant pas de spécifier explicitement de clause "select" dans certains cas.

```
private ProjectJPA2 retrieveTheProjectInDatastore(String argname) throws
    FailedOperation {
```

[28] http://docs.oracle.com/javaee/7/api/javax/persistence/criteria/CriteriaBuilder.html

```
    assert argname != null;

    try {
        final EntityManager em = getEntityManager();
        final CriteriaBuilder qb = em.getCriteriaBuilder();
        final CriteriaQuery<ProjectJPA2> criteria =
           qb.createQuery(ProjectJPA2.class);
        final Root<ProjectJPA2> r = criteria.from(ProjectJPA2.class);
        final Predicate condition = qb.equal(r.get("name"), argname);
        final TypedQuery<ProjectJPA2> q =
           em.createQuery(criteria.where(condition));

        final Iterator<ProjectJPA2> iter = q.getResultList().iterator();
        if (iter.hasNext()) {
            return iter.next();
        }

        return null;
    } catch (final Exception e) {
        throw new FailedOperation("Exception thrown during retrieval of
           Extent.", e);
    }
}
```

Listing 8 – Exemple d'utilisation de l'API CriteriaBuilder dans l'implémentation OpenJPA

```
private ProjectODB retrieveTheProjectInDatastore(String argname) throws
   FailedOperation {
    assert argname != null;

    try {
        final EntityManager em = getEntityManager();
        final CriteriaBuilder qb = em.getCriteriaBuilder();
        final CriteriaQuery<ProjectODB> criteria =
           qb.createQuery(ProjectODB.class);
        final Root<ProjectODB> r = criteria.from(ProjectODB.class);
        final Predicate condition = qb.equal(r.get("name"), argname);
        criteria.where(condition);
        criteria.select(r);

        final TypedQuery<ProjectODB> q = em.createQuery(criteria);

        final Iterator<ProjectODB> iter = q.getResultList().iterator();
        if (iter.hasNext()) {
            return iter.next();
        }

        return null;
    } catch (final Exception e) {
        throw new FailedOperation("Exception thrown during retrieval of
           Extent.", e);
    }
}
```

Listing 9 – Exemple d'utilisation de l'API CriteriaBuilder dans l'implémentation ObjectDB

4.4.4 Mise en place du enhancement des classes

La résolution des problèmes rencontrés avec les mappings JPA, a permis de faire fonctionner l'application MetaDone avec la nouvelle implémentation du metarepository. Afin d'assurer des performances optimales, il a été nécessaire de configurer l'*enhancement* des classes, comme recommandé par la documentation d'ObjectDB[29] (bien que vivement encoragée, cette configuration reste optionnelle).

À cette fin, la librairie ObjectDB fournit un exécutable qui applique des modifications sur le bytecode des entités compilées. Il est possible d'utiliser cet outil soit en phase de post-compilation soit au moment du chargement des classes dans la JVM (via un Java agent). Cependant l'*enhancement* au chargement n'est pas recommandé en dehors de la phase de développement.

Selon la documentation, ces modifications du bytecode améliorent l'efficacité de la gestion des entités de trois manières :

- Un suivi efficace des modifications des champs persistants, évitant ainsi la nécessité de comparer la représentation en base de données et l'entité de l'application. Il s'agit donc d'ajouter du code aux classes d'entité afin d'automatiquement notifier ObjectDB de chaque modification d'un champ persisté.
- Le *lazy loading* pour les relations one-to-one. En effet, sans modification du bytecode, le comportement par défaut, pour ce type de relation est le chargement immédiat (mode *eager*) à contrario du comportement pour les relations de type One-to-Many.
- L'ajout de méthodes optimisées afin de remplacer l'utilisation de la réflexion. En effet l'utilisation de la réflexion peut s'avérer coûteuse en terme de performance.

Dans le listing 16 présenté en annexe, nous trouvons un exemple de modifications du bytecode pour la classe Facets de notre metarepository (le code source a été reconstitué à l'aide d'un outil de décompilation FernFlower), .

L'inconvénient de l'outil fourni par ObjectDB est que lors de son exécution, il tente d'explorer le code provenant d'autres bundles que celui contenant l'implémentation du repository, afin d'y appliquer des modifications si nécessaire. En pratique, il tente d'accéder aux interfaces du repository définies dans metadone_bundle_repository, ainsi qu'aux exceptions utilisées provenant de metadone_common. Ce que ne permet pas la structuration du projet MetaDone, telle qu'elle était. Étant donné que c'est trois bundles ne nécessitaient pas le même niveau de modularité que les plugins par exemple, il a dès lors été possible de restructurer le projet pour regrouper ces modules en un seul bundle (bundle core).

Cependant le fait de coupler l'implémentation ObjectDB du metarepository avec le contenu des autres bundles n'était pas une option totalement satisfaisante car pour pouvoir tester les deux implémentations, cela aurait nécessité d'introduire de la duplication (un bundle core par implémentation). Loin d'être un énorme problème, cette constatation a représenté une

[29] `http://www.objectdb.com/java/jpa/tool/enhancer`

opportunité de mener une reflexion sur la place d'OSGI dans le projet MetaDone. En a résulté une proposition de solution consistant à séparer le code métier de MetaDone du code et de la configuration spécifique à l'exécution dans un environnement OSGI.

Une première phase à consister simplement à relocaliser le code métier contenu dans les bundles vers des projets séparés, tout en conservant leur structure. Afin de faciliter, la construction de ces projets sous forme de jar utilisable par les bundles correspondant, il a été décidé d'utiliser Maven[30]. Maven possède plusieurs avantages intéressants pour le projet MetaDone, comme la gestion des dépendances (librairies) et l'automatisation des tests unitaires.

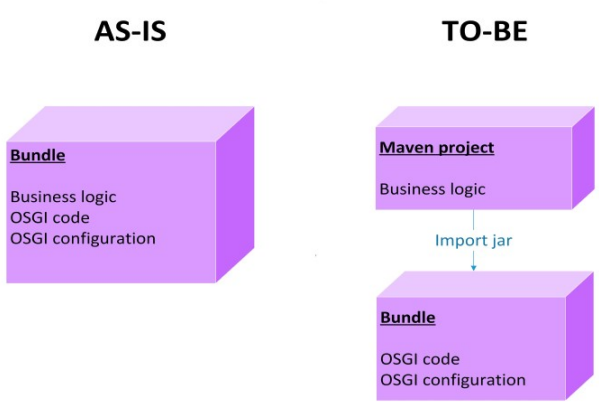

FIGURE 6 – Première phase de restructuration des bundles

Les bundles OSGI quant à eux restent des projets Eclipse. Ils possèdent la configuration nécessaire à l'exécution et importent dorénavant le code métier sous forme de jar (jar contenant l'ensemble des dépendances exportées par le bundle mais sans pour autant contenir le code métier des autres bundles car ceux-ci sont fournis par l'environnement d'exécution).

Une fois cette modification réalisée et testée sur les bundles (common, repository, jpa2 et odb), il était possible de les fusionner, plus aisément, en un seul bundle core. Le changement d'implémentation de technologies de metarepository consiste alors à importer le jar correspondant à l'une ou l'autre implémentation dans ce bundle core (copie de jar dans le répertoire lib du bundle correspondant réalisée par une tâche Maven).

4.4.5 Démarrage de l'application MetaDone

La dernière phase a porté sur la mise en place d'un mécanisme simple pour pouvoir tester facilement les deux implémentations (OpenJPA/h2 et ObjectDB). Le mécanisme concernant le bundle core a déjà été mentionné. Il restait le travail à effectuer pour permettre de démarrer le client (metadone_bundle_client) en utilisant l'une ou l'autre implémentation.

Les propriétés nécessaires à la configuration d'une connection pour la base de données H2 et ObjectDB n'étant pas exactement les mêmes, il a été nécessaire de permettre la manipulation

[30] `http://maven.apache.org/`

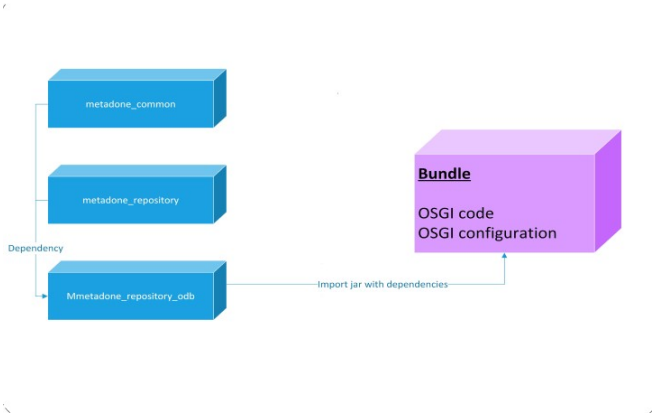

FIGURE 7 – Structure finale du bundle core

de fichier de propriétés différent en fonction de l'implémentation utilisée. Ceci est réalisé par la mise en place d'un mécanisme basé sur l'utilisation d'une variable d'environnement (*repository.type*). Cette variable d'environnement est employée par la classe *GlobalParameters* pour déterminer le nom du fichier de propriétés à exploiter.

```
package metadone.client;

public final class GlobalParameters {
    /**
     * relative path of the file with the default parameters to
        initialize the user's preferences
     */
    static final public String DefaultPropertiesFile=
       String.format("/%s.properties",
       System.getProperty("repository.type"));
    /**
     * the name of the file in the home directory that stores the
        preferences
     */
    static final public String
       PreferencesFileNameInHomeDirectory="metadone.prefs";

}
```

Listing 10 – Classe *GlobalParameters*

La valeur de cette variable d'environnement est ensuite configurée dans le fichier *.product* permettant de lancer l'application MetaDone. Deux fichiers *.product* ont dès lors été créés. L'un destiné au lancement de MetaDone en utilisant le metarepository OpenJPA/H2 et l'autre utilisant l'implémentation ObjectDB.

5 Évaluation des performances des deux implémentations

5.1 Méthodologie

Bien que l'utilisation de la nouvelle implémentation semblait améliorer l'utilisabilité de MetaDone en réduisant les temps de latence lors d'opérations (illustré par moins de temps de latence pour la création et le chargement du métamodèle BPMN par exemple), il était bien entendu nécessaire de pouvoir objectiver les gains de performance de la nouvelle implémentation par rapport à l'ancienne.

Dans cette optique, l'approche méthodologique a été inspirée par celle utilisé afin d'évaluer les performances et la scalabilité de la couche de persistance d'EMF présentée dans la section 3. Cette approche se basait sur l'utilisation de scénarios représentatifs de cas d'utilisation des outils MDE. Ces scénarios incluaient la création de modèles de tailles variés, l'accès à ces modèles et enfin l'exécution de requêtes sur ces modèles. Les données des modèles utilisés étaient tirés du workshop GraBaTs [Gra].

En suivant le même type d'approche, il paraissait pertinent d'établir des scénarios représentatif des problèmes de performance de l'implémentation actuelle, et suffisament proche des cas d'utilisation courant de MetaDone. Les scénarios qui ont été identifiés étaient la création massive d'objet, la mise à jour massive d'objet, la suppression massive d'objet, ainsi que l'accès à de nombreux objets. Il s'agit de scénarios qui sont, par exemple, représentatifs d'une utilisation de type *Extract Transform Load*.

Parmi ces scénarios, un seul scénario a été retenu. Il s'agit du type consistant à la création massive d'objets. Les motivations de ce choix sont doubles. D'une part, il s'agit du scénario qui peut être considéré comme l'un des scénarios les plus complexes. La création est complexe car elle implique la sérialisation de l'entièreté d'un graphe d'objet en base de données (ce qui n'est pas forcément le cas par exemple pour une mise à jour). La deuxième raison est qu'il est également représentatif d'autres cas d'utilisation que ceux du type ETL. En effet, les problèmes de performance rencontrés lors de la création de métamodèles et de modèles se manifestent dans MetaDone pour bon nombre d'activité de modèlisation.

À partir de ce scénario, il a été défini trois sous scénarios afin de varier les types d'objets créés dans le metarepository et de permettre d'évaluer la scalabilité :

- Enchainement de création de métamodèles :

  `for (1..100) do { create MetaModel in metarepository }`

- Enchainement de création de modèles pour un métamodèle :

  `for (1..100) do { create Model of MetaModel in metarepository}`

- Mesure de scalabilité pour la création des modèles :

  `for (1..100) do { create Model of size i*K in metarepository }`

Afin de s'abstraire des éléments non essentiels (tels que la partie GUI), il semblait intéressant d'évaluer ces scénarios en invoquant directement les méthodes provenant de l'API business de MetaDone (située dans le bundle metadone_bundle_metabusiness).

5.2 Métamodèle et modèles utilisés

5.2.1 Présentation

Pour la réalisation de ces scénarios, il a été nécessaire de faire un choix de métamodèle à utiliser. Le métamodèle représentant les réseaux de Petri a été choisi car il était à la fois simple à appréhender et permettait également d'être aisément complexifié.

Les réseaux de Petri sont des outils graphiques et mathématiques permettant de représenter des systèmes dynamiques qui évoluent d'un état à un autre lorsque des événenements surviennent. Ils ont été proposés par Carl Adam Petri dans sa thèse de doctorat en 1962. Ils sont représentés sous forme de graphes biparti avec d'une part des noeuds représentant les places et d'autres part des noeuds représentant les transitions. Les places représentent les états du systèmes et les transitions représentent des événements. Les places peuvent contenir des jetons (resources), chaque franchissement de transition transfert un ou plusieurs jetons d'une place à une autre.

Le code du listing 11 illustre la manière de créer ce métamodèle dans MetaDone, nous retrouvons également son diagramme de classe correspondant à la figure 8.

```
MetaModel petri = workspace.getMainModel().createMetaModel("Petri");
MetaObject place = petri.createMetaObject("Place");

place.createMetaProperty(String.class, "Place.name", 1, petri);
place.createMetaProperty(Long.class, "Place.count", 1, petri);

MetaObject transition = petri.createMetaObject("Transition");

petri.createMetaRole("Source", MetaRole.Cardinality.MANY_TO_MANY, place,
    transition);
petri.createMetaRole("Target", MetaRole.Cardinality.MANY_TO_MANY,
    transition, place);
```

Listing 11 – Implémentation de la création du métamodèle de Petri

FIGURE 8 – Métamodèle de réseau de Petri

À partir de ce métamodèle nous pouvons ensuite créer une instance de celui-ci. Le code d'une instance simplifié est présenter au listing 12, et la visualisation de celui-ci dans MetaDone à la figure 9.

```
ConcreteModel mynet = workspace.getMainModel().createModel(petri);

ConcreteObject p1 = mynet.createObject(place);
p1.createProperty(count, 2L, mynet);
p1.createProperty(name, "P1", mynet);

ConcreteObject p2 = mynet.createObject(place);
p2.createProperty(name, "P2", mynet);

ConcreteObject t1 = mynet.createObject(transition);
mynet.createRole(source, p1, t1);
mynet.createRole(target, t1, p2);
```

Listing 12 – Implémentation de la création d'un modèle de Petri

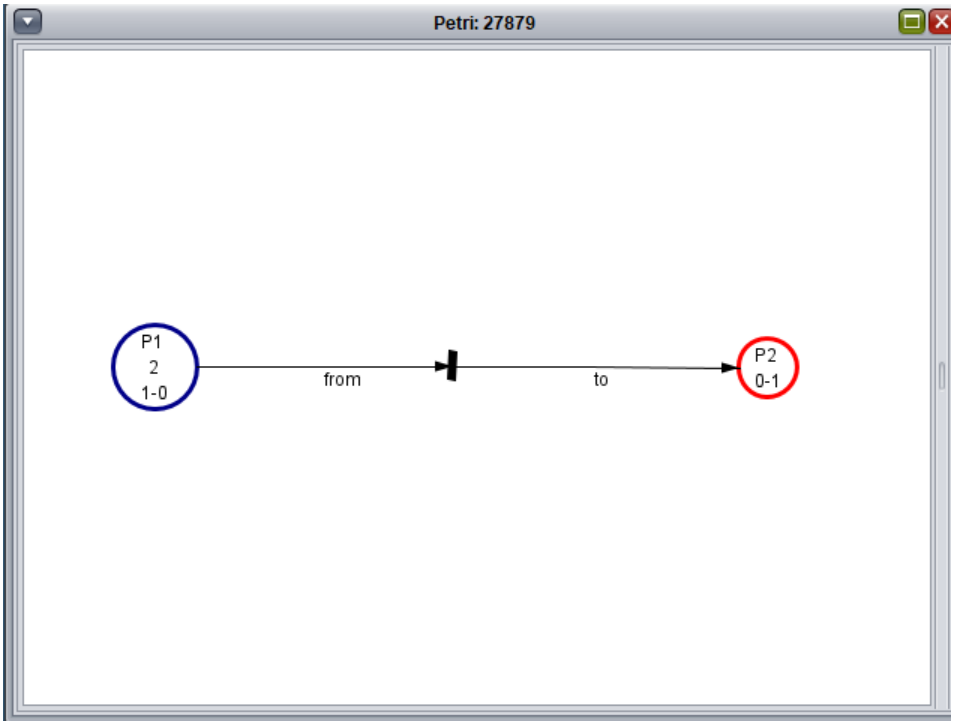

FIGURE 9 – Visualisation dans MetaDone d'une instance du métamodèle de Petri

Pour ce qui concerne l'évaluation de la scalabilité, une première approche *naive* a été utilisée. Elle consistait simplement à construire de façon itérative un modèle similaire au modèle

précédent. En ajoutant à chaque itération une place et une transition destinée à lier la place créée dans l'itération courante avec celle de l'itération précédente.

```
ConcreteModel mynet = workspace.getMainModel().createModel(petri);

ConcreteObject previousObject = null;
for (int i = 0; i < size; ++i) {
    ConcreteObject p = mynet.createObject(place);
    p.createProperty(count, 2L, mynet);
    p.createProperty(name, "P" + i + identifier, mynet);

    if (previousObject != null) {
        ConcreteObject t = mynet.createObject(transition);
        mynet.createRole(source, previousObject, t);
        mynet.createRole(target, t, p);
    }
    previousObject = p;
}
```

Listing 13 – Implémentation de la création d'un modèle de Petri de K objets

L'évaluation du temps d'exécution des scénarios avec ce métamodèle et ce modèle ont déjà permis d'illustrer le gain de performance de l'implémentation du metarepository basée sur ObjectDB (voir section 5.3). Néanmoins, la complexité des métamodèles et des modèles pourrait influencer négativement les performances de l'une ou l'autre des implémentations (pourrait impliquer un plus grand nombre de liens). Il semblait dès lors judicieux de tenter de complexifier le métamodèle ainsi que les modèles s'y rapportant.

5.2.2 Complexification du métamodèle

La complexification du métamodèle a consisté à introduire l'utilisation de la notion d'héritage pour l'étendre. Afin de rester cohérent avec les réseaux de Petri, les inspirations pour l'utilisation de l'héritage, ont été trouvées dans les extensions existantes des réseaux de Petri. En effet, la spécialisation des places s'inspire des réseaux de Petri P-temporisé [Sif77]. Quant à l'extension des transitions, elle s'inspire de l'idée de pouvoir associer un coût au transition (comme utilisé dans [LRH06]).

La création du métamodèle enrichi est illustrée par le code présenté dans le listing 14 et nous retrouvons sa représentation sous forme de diagramme de classe à la figure 10

```
MetaModel petri = mainMetaModel.createMetaModel("Petri");
MetaObject place = petri.createMetaObject("Place");

place.createMetaProperty(String.class, "Place.name", 1, petri);
place.createMetaProperty(Long.class, "Place.count", 1, petri);

MetaObject timedPlace = petri.createMetaObject("TimedPlace", place);

timedPlace.createMetaProperty(Long.class, "TimedPlace.time", 1, petri);

MetaObject transition = petri.createMetaObject("Transition");
```

```
petri.createMetaRole("Source", MetaRole.Cardinality.MANY_TO_MANY, place,
    transition);
petri.createMetaRole("Target", MetaRole.Cardinality.MANY_TO_MANY,
    transition, place);

MetaObject transitionWithCost =
    petri.createMetaObject("TransitionWithCost", transition);

transitionWithCost.createMetaProperty(Long.class,
    "TransitionWithCost.cost", 1, petri);
```

Listing 14 – Implémentation de la création du métamodèle de Petri étendu

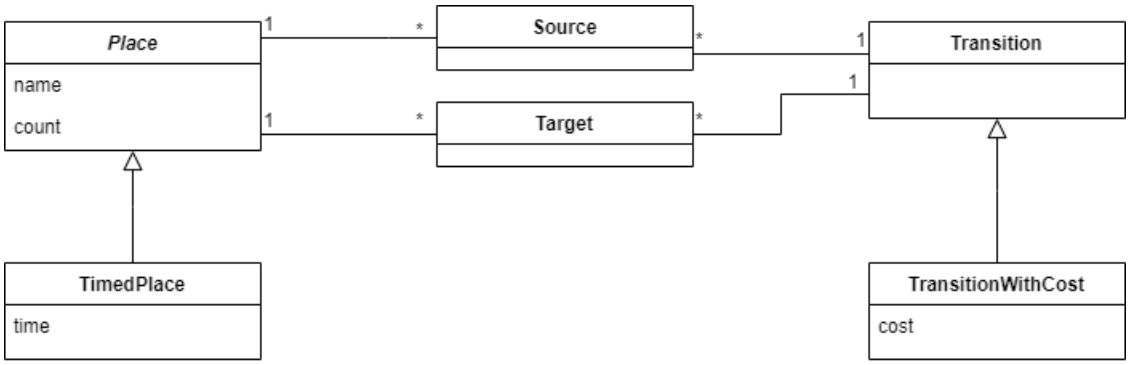

FIGURE 10 – Métamodèle étendu de réseau de Petri

Le meta objet *TimedPlace* est donc destiné à représenter une place possédant une durée minimale en seconde de présence d'un jeton dans celle-ci. Il s'agit donc d'une spécialisation du meta objet *Place*. Le meta objet *TransitionWithCost* quant à lui représente une spécialisation du meta objet *Transition* et qui permet de représenter une transition dont on peut spécifier le nombre de jeton consommé lors de son franchissement.

5.2.3 Complexification des modèles

Il paraissait également intéressant de pouvoir générer des modèles de manière moins *naive* afin de représenter des modèles plus proches de modèles de Petri réels. C'est-à-dire de pouvoir générer non seulement des modèles exploitant les extensions du métamodèle de Petri étendu présentés dans la section 5.2.2 mais également des modèles plus denses.

La nécessité d'implémenter manuellement la création des métamodèles et des modèles, représentait un obstacle pour l'implémentation de modèle plus complexe. En effet, alors que cette tâche reste simple pour des métamodèles ou des modèles de petite taille, elle devient peu praticable pour des modèles possédant une ou plusieurs centaines d'objets. Afin donc de faciliter l'implémentation de modèles destinés à la mesure de scalabilité, un outil de génération du code de création de modèles a été réalisé.

Cet outil permet de générer un fichier source contenant le code de création d'un modèle de Petri. La configuration d'un modèle pour une taille donnée et un ratio transition/place est

généré de manière aléatoire. Le code source est généré à partir d'un template en utilisant un framework de templating pour le monde java Freemarker[31].

La génération du modèle à transposer dans le template est réalisé en partant de la possibilité de définir un réseau de Petri sous forme de matrices d'incidence. L'idée est donc de définir notre réseau de Petri en générant deux matrices :

- Une matrice de Pré-incidence : Matrice à n lignes et m colonnes avec n le nombre de places et m le nombre de transitions. Dans la littérature, chaque élément de cette matrice $Pre(P_i, T_j)$ correspond au nombre de jetons à enlever dans P_i en franchissant T_j. Cependant, dans notre cas il s'agira simplement de définir si un arc orienté de P_i à T_j existe.
- Une matrice de Post-incidence : Matrice à n lignes et m colonnes avec n le nombre de places et m le nombre de transitions. Dans la littérature, chaque élément de cette matrice $Post(P_i, T_j)$ correspond au nombre de jetons à rajouter dans P_i en franchissant Tj. Cependant, dans notre cas il s'agira simplement de définir si un arc orienté de T_j à P_i existe.

La génération de la configuration du modèle consiste donc à remplir de manière aléatoire les deux matrices d'incidences (valeur 1 ou 0, indiquant la présence ou l'absence d'un arc entre P_i et T_j). Pour éviter les cycles entre une place et une transition, le générateur ne permet pas d'avoir d'intersection entre ces deux matrices. Le générateur s'assure également que chaque place et chaque transition soient connectées au graphe.

Une fois générées ces deux matrices servent donc à compléter le template afin de générer le code source de la création du modèle. En ce qui concerne le modèle utilisant le métamodèle étendu, le choix entre super-type et sous-type se fait également de manière aléatoire.

La visualisation d'un modèle généré à l'aide de l'outil est présenté à la figure 11 (le code est présenté en annexe dans le listing 17).

[31] `http://freemarker.apache.org/`

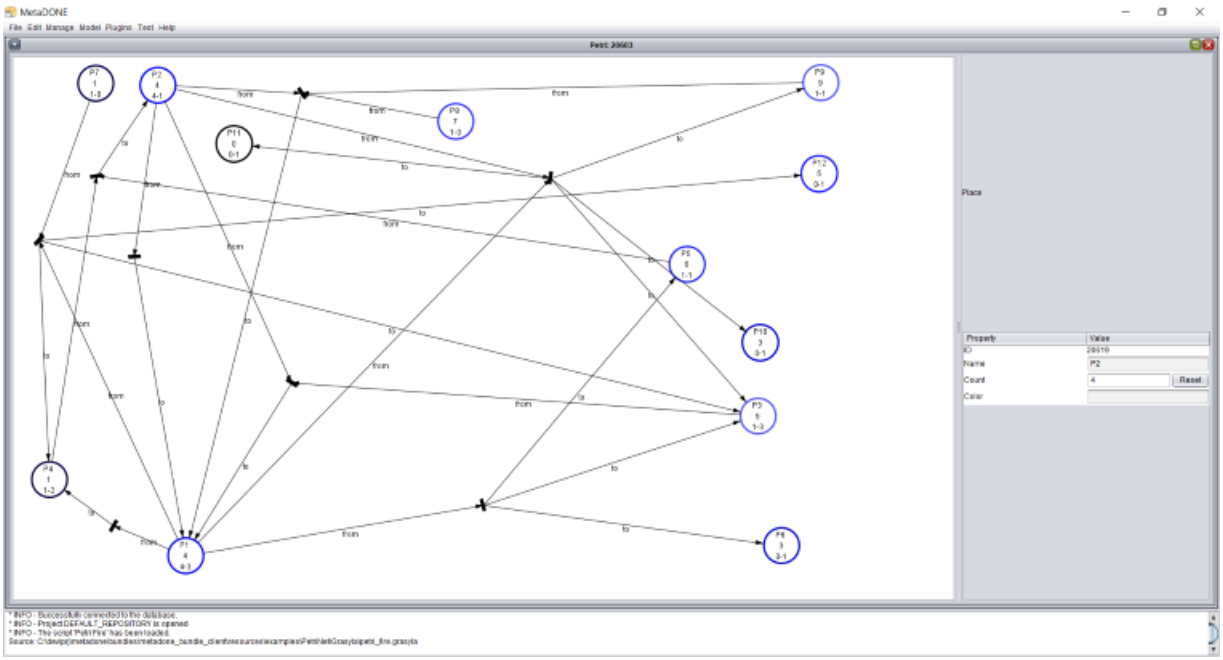

FIGURE 11 – Visualisation du modèle généré dans MetaDone

5.3 Benchmarking

5.3.1 Implémentation

L'implémentation du benchmarking a simplement consisté à mettre en place un microframework permettant de mesurer le temps d'exécution des opérations de création massive de métamodèles/modèles. Celui-ci réutilise la classe utilitaire *Chrono* présente dans MetaDone.

```
public BenchmarkResult bench(String operationType, String modelType,
    Runnable runnable) {
    Chrono result = bench(modelType + " " + operationType, runnable);

    return new BenchmarkResult(operationType, modelType, result);
}

private static Chrono bench(String name, Runnable runnable) {
    log.info("Running benchmark: " + name);

    Chrono chrono = new Chrono();
    chrono.start();

    runnable.run();

    chrono.pause();

    log.info("Time elapsed for " + name + ": " +
        chrono.getElapsedMilliSeconds());

    return chrono;
```

```
}
```

Listing 15 – Extrait du microframework mis en place pour le benchmarking

Les instances de *BenchmarkResult* sont ensuite destinées à être traitées afin de soit afficher les résultats dans la console soit de produire un csv.

Il s'agit, bien entendu, d'une méthode d'évaluation rudimentaire. Les travaux futures pourraient incorporer d'autres types de métriques (throughput, temps moyen, ...) mais également utiliser une approche moins naive avec un outil tel que le framework JMH[32].

5.3.2 Résultats

L'ensemble des benchmarks ont été éxécutés sur un processeur Intel Core I5-8350U à 1.70GHz avec 16GB de RAM avec comme OS Windows 10 Enterprise, comme JVM la version 1.8.0_191-x64 et comme base de données la version 1.4.182 de H2 et la version 2.8.2 d'ObjectDB.

Afin d'isoler au mieux les évaluations mais également de contourner les limitations de la base de données ObjectDB concernant le nombre d'objets persistés, chaque scénario a été exécuté sur une base de données vide.

5.3.2.1 Création des métamodèles

Le premier scénario évalué a donc été la création de 100 métamodèles de type Réseau de Petri (métamodèle et métamodèle étendu).

Métamodèle	OpenJPA/H2 (en ms)	ObjectDB (en ms)
Réseaux de Petri	10172	1288
Réseaux de Petri étendu	16296	1343

Table 1 – Temps d'exécution pour la création massive de métamodèles

[32] `http://openjdk.java.net/projects/code-tools/jmh/`

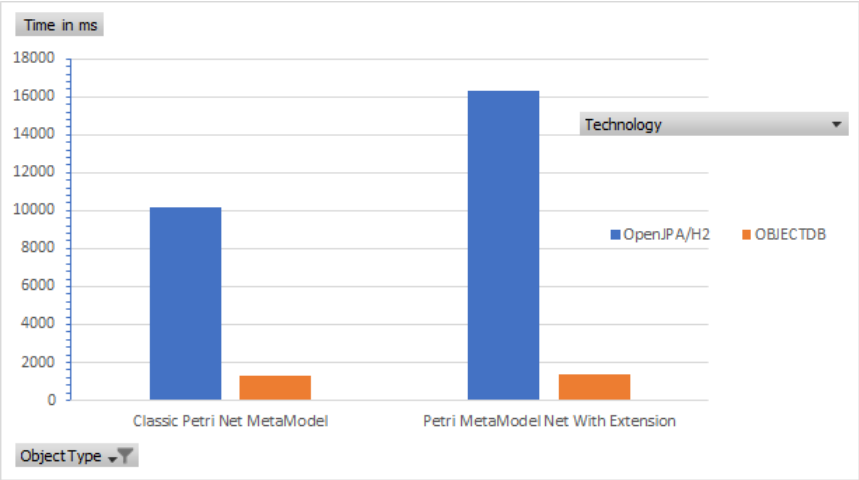

FIGURE 12 – Résultats du benchmarking pourla création des métamodèles

Une première constatation est qu'au delà de la différence frappante d'ordre de grandeur pour les temps d'exécution entre les deux implémentations, la complexité du métamodèle impacte de manière plus marquante l'implémentation OpenJPA/H2.

5.3.2.2 Création des modèles

Les résultats de cette section concerne non seulement les performances mais également la scalabilité pour la création de modèles. Plusieurs types de modèles sont évalués :

- Les réseaux de Petri créés de façon *naive*

Taille	OpenJPA/H2 (en ms)	ObjectDB (en ms)
10	12047	238
20	22520	482
40	45423	847
80	93099	2812
160	193705	6357

TABLE 2 – Temps d'exécution pour la création massive de modèles de Petri

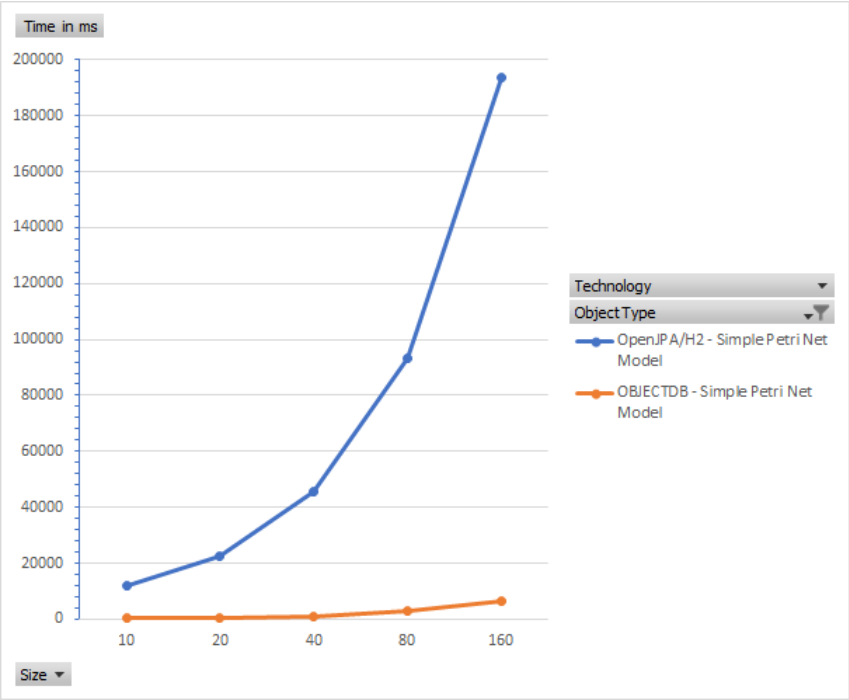

FIGURE 13 – Résultats pour la création de modèles de Petri

- Les réseaux de Petri générés aléatoirement et utilisant le métamodèle étendu

Taille	OpenJPA/H2 (en ms)	ObjectDB (en ms)
10	10440	376
20	19498	559
40	35962	820
80	76293	2260
160	166323	5311

TABLE 3 – Temps d'exécution pour la création massive de modèles de Petri étendus

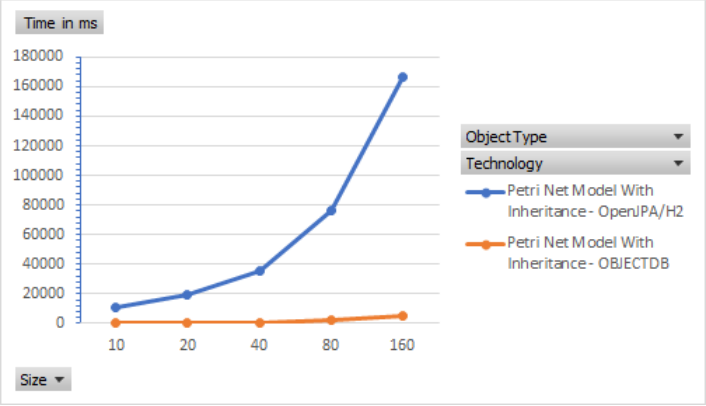

FIGURE 14 – Résultats pour la création de modèles de Petri étendus

- Les réseaux de Petri *dense*

Taille	OpenJPA/H2 (en ms)	ObjectDB (en ms)
10	7447	135
20	14379	387
40	29133	663
80	60289	2550
160	128115	3448

TABLE 4 – Temps d'exécution pour la création massive de modèles de Petri denses

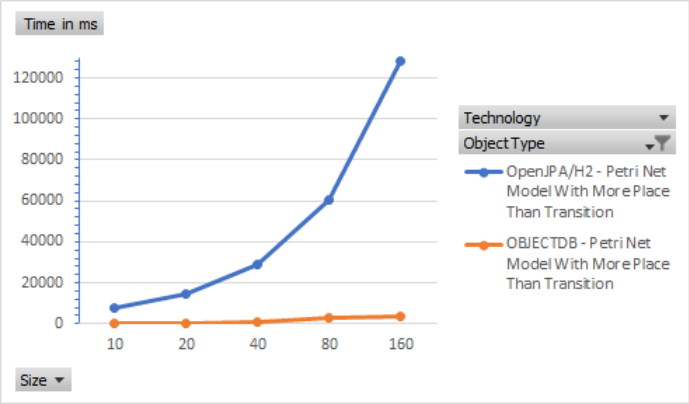

FIGURE 15 – Résultats pour la création de modèles de Petri denses

5.3.3 Discussion

L'évaluation des performances des deux implémentations a suivi une approche méthodologique inspirée de celle suivie pour l'évaluation des solutions proposées pour EMF (GraBaTs). Un certain nombre de scénarios considérés comme difficiles ont été sélectionnés pour mener à bien cette évaluation comparative. Cette phase d'évaluation nous permet de tirer certaines conclusions.

Nous pouvons constater que sur l'évaluation de l'exécution des différents scénarios, l'implémentation utilisant ObjectDB offrait de bien meilleurs résultats en terme de performance que l'implentation basée sur OpenJPA/H2. Ce qui est conforme à nos attentes. Il est important de noter que sur les différents scénarios évalués, l'influence de la complexité des modèles avait été marginale sur les performances (ce qui est vrai pour les deux implémentations).

Comme mentionné, cette phase a consisté à n'évaluer qu'un sous-ensemble des scénarios représentatifs de cas d'utilisation courants de MetaDone. Celle-ci a néanmoins permis de dresser un premier bilan positif concernant la performance de la solution proposée.

6 Conclusion

6.1 Contribution

Une solution prometteuse afin de résoudre les problèmes de performance du metarepository de MetaDone a été proposée et évaluée. Cette solution a été inspirée par les travaux menés dans le cadre de l'amélioration de la performance et de la scalabilité du repository d'EMF (travaux présentés dans l'état de l'art à la section 3).

Bon nombre de ces travaux ont mis l'accent sur l'utilisation d'un paradigme alternatif au modèle relationnel pour répondre efficacement au besoin de performance et de scalabilité du *Model Driven Engineering*. Cette constatation a orienté le choix d'un changement de paradigme pour l'implémentation du metarepository pour répondre au besoin de performance.

Même si la solution implémentée a été inspirée par ces recherches, elle a néanmoins permis d'apporter une contribution différente. En effet, alors que les travaux menés sur EMF ont porté sur l'utilisation de différentes bases de données NoSQL, aucun de ces travaux n'avait présenté de solution fondée sur une base de données orientée objet.

Nous avons vu que les bases de données orientées objet possédaient certains avantages similaires à ceux offerts par les bases de données orientées graphe (modèle et *index free adjacency*). Avantages dont il était possible de tirer parti dans l'implémentation du metarepository afin de répondre aux besoins de performance et de scalabilité.

Bien que les performances n'aient été évaluées que sur un petit nombre de scénarios (considérés comme les plus importants), la nouvelle implémentation semble répondre aux attentes en cette matières.

6.2 Approche méthodologique pour l'évaluation des performances

Une fois l'intégration d'ObjectDB dans MetaDone, il était important d'objectiver les gains de performance de la nouvelle solution par rapport à l'ancienne implémentation. Dans cette optique, il était important de suivre une approche méthodologique ayant fait ces preuves dans un domaine proche. C'est pourquoi là aussi l'inspiration est venue de l'approche utilisée pout évaluer les performances de la couche de persistance des outils MDE.

En effet, à l'instar de ce qui a été réalisé pour l'évaluation des méthodes MDE avec GraBaTs, la méthode suivie a consisté à définir des scénarios suffisamment représentatifs des cas d'utilisation courant de MetaDone. Parmi les scénarios envisagés, nous n'en avons retenu qu'un petit nombre représentant les cas considérés comme les plus difficiles. Ensuite à partir des scénarios types sélectionnés, construire les métamodèles et modèles utilisés pour disposer d'un jeu de données représentatif. Ces jeux de données ont permis d'évaluer les performances et également dans une certaine mesure la scalabilité des deux solutions.

Les conclusions de l'évaluation des scénarios séléctionnés, ont permis de montrer que l'implémentation du metarepository utilisant ObjectDB, semble être une solution qui répond aux attentes en matière de performance.

Bien que l'approche suivie se soit révélée pertinente, nous pourrions malgré tout adresser une critique. Il aurait peut-être été intéressant de mener ce travail en inversant les étapes de l'approche utilisée. Effectivement, ce travail à consisté à implémenter une solution choisie aux problèmes de performance du metarepository, avant de mener une évalution comparative des deux implémentations. Il était tout à fait possible de procéder d'une autre manière en élaborant dans une première phase les scénarios d'évaluation, pour objectiver tout d'abord les problèmes de performance du metarepository et en déduire des scénarios d'attributs de qualité à manière de [BCK12]. Ces scénarios d'attributs de qualité auraient permis, après l'implémentation d'une solution et de son évaluation, de définir si elle répondait aux attentes. Cette approche est d'ailleurs l'approche généralement suivie dans l'industrie.

6.3 Critique d'ObjectDB

La nouvelle implémentation du metarepository se base sur l'utilisation d'ObjectDB qui a été présentée dans la section4.1. Après évaluation, cette implémentation semble répondre aux attentes, ce qui représente un point positif non négligable.

Cependant cette solution n'est pas exempte de défauts. Certains de ces défauts ont été relevés lors de la résolution des problèmes de mapping. En effet, on constate que l'implémentation de JPA d'ObjectDB ne suit clairement pas l'évolution de la spécification. Nous avons vu que cela impose parfois d'utiliser des contructions complexe que justement l'évolution de la spécification avait pour objectif de faire disparaître. Il est à noter que ce point négatif est à relativiser. En effet, cet aspect n'impacte que la phase de développement et le problème existe également pour d'autres implémentations de JPA.

La version utilisée pour l'implémentation était une version sous licence gratuite, dont les autorisations d'utilisation sont assez large (y compris commerciale). Néanmoins un défaut important a été découvert durant la phase d'évaluation de la scalabilité. L'inconvénient de cette version est qu'elle possède deux limitations. La limitation concernant le nombre de classe d'entité (10) cependant cet aspect n'impacte pas son utilisation dans MetaDone. Par contre, la limitation d'un million d'objets persistés est un inconvénient important. Cet inconvénient a d'ailleurs été un frein pour envisager la possibilité d'évaluation de scénarios avec un ordre de grandeur supérieur à ceux présentés dans la section évaluation 5.3.

A l'avenir, si l'implémentation est retenue pour une utilisation dans l'outils MetaDone, il sera nécessaire d'envisager d'utiliser une autre version d'ObjectDB (sous licence payante).

6.4 Perspectives

Un certain nombre d'éléments ont été laissés en suspens dans ce travail. Certains d'entre-eux paraissent néanmoins suffisamment intéressant que pour faire l'objet de travaux ultérieurs. Parmi ceux-ci nous trouvons les éléments suivants :

- L'évaluation comparative à l'aide d'autres scénarios : En effet, seul certains scénarios ont été évalués. Afin de s'assurer de manière plus certaine de l'adéquation de la nouvelle implémentation du metarepository, il serait intéressant de réaliser l'évaluation des scénarios laissés en suspens dans ce travail.

- Implémentation de l'évaluation : Alors que l'évaluation a été réalisée de manière rudimentaire, en implémentant un genre de microframework, il pourrait être intéressant d'utiliser, pour ce faire, un framework plus élaboré tel que JMH[33] par exemple.

- Correction des problèmes de performance liés à la conception de l'API : Il s'agit d'une piste d'amélioration qui a été laissée en suspens, mais qui semble également prometteuse y compris pour la nouvelle implémentation. Pour mener à bien cette partie, il pourrait être intéressant de suivre une méthodologie spécifique d'optimisation de performance de code, en utilisant certains outils d'analyse tel que JClarity[34] ou JProfiler[35] pour aider à mettre en lumière certains de ces problèmes.

- L'utilisation d'une base de données orientée Graphe ou orientée Map : Il s'agit de deux approches qui n'ont pas été retenues, néanmoins chacune d'entre-elle pourrait s'avérer intéressante. De plus, l'étude de ces éventuelles implémentations pourraient, dorénavant, être comparée à l'approche orientée objet suivie dans ce travail.

- Construire MetaDone au-dessus d'EMF : Cette alternative n'a pas été retenue car elle paraissait trop ambitieuse. Cette solution permettrait de bénéficier des nombreuses avancées issues des recherches menées sur le monde EMF. Néanmoins, cette solution paraît un chantier important et pourrait avoir un rapport coût/bénéfice peu avantageux par rapport à l'exploration des deux précédents points.

[33] `http://openjdk.java.net/projects/code-tools/jmh/`
[34] `http://www.jclarity.com/`
[35] `http://www.ej-technologies.com/products/jprofiler/overview.html`

A Annexes

```
package metadone.repository.kernel.odb;

import com.objectdb.spi.OMember;
import com.objectdb.spi.OReader;
import com.objectdb.spi.OType;
import com.objectdb.spi.OWriter;
import com.objectdb.spi.TrackableUserType;
import com.objectdb.spi.Tracker;
import java.util.Collection;
import java.util.EnumSet;
import java.util.Iterator;
import javax.persistence.Basic;
import javax.persistence.Embeddable;
import javax.persistence.PostLoad;
import javax.persistence.Transient;
import metadone.repository.kernel.FacetType;

@Embeddable
public class Facets implements TrackableUserType {
    @Transient
    private final EnumSet<FacetType> facets =
       EnumSet.noneOf(FacetType.class);
    @Basic
    private int dataObjectFacets;

    public Facets() {
    }

    public String toString() {
        return "Facets" + this.facets.toString();
    }

    public EnumSet<FacetType> getFacets() {
        return this.facets;
    }

    public void addFacet(FacetType facetType) {
        this.facets.add(facetType);
        __odbSet_dataObjectFacets(this, toInt(this.facets));
    }

    public void addAllFacets(Collection<FacetType> facetTypes) {
        this.facets.addAll(facetTypes);
        __odbSet_dataObjectFacets(this, toInt(this.facets));
    }

    public void removeAllFacets(Collection<FacetType> facetTypes) {
        this.facets.removeAll(facetTypes);
        __odbSet_dataObjectFacets(this, toInt(this.facets));
    }
```

```
@PostLoad
void loadFacets() {
    this.facets.addAll(toFacets(__odbGet_dataObjectFacets(this)));
}

static int toInt(EnumSet<FacetType> facets) {
    int result = 0;

    FacetType t;
    for(Iterator var2 = facets.iterator(); var2.hasNext(); result |=
       1 << t.ordinal()) {
        t = (FacetType)var2.next();
    }

    return result;
}

static EnumSet<FacetType> toFacets(int types) {
    EnumSet<FacetType> result = EnumSet.noneOf(FacetType.class);
    FacetType[] values = FacetType.values();

    for(byte i = 0; types != 0; types >>= 1) {
        if ((types & 1) != 0) {
            FacetType t = values[i];
            result.add(t);
        }

        ++i;
    }

    return result;
}

public void __odbSetTracker(Tracker var1) {
    this.__odbTracker = var1;
}

public Tracker __odbGetTracker() {
    return this.__odbTracker;
}

public TrackableUserType __odbNewInstance() {
    return new Facets();
}

public static void __odbSet_facets(Facets var0, EnumSet var1) {
    if (var0.__odbTracker != null &&
       var0.__odbTracker.beforeModifyMember(0)) {
        var0.facets = var1;
        var0.__odbTracker.afterModify();
    } else {
        var0.facets = var1;
    }
```

```
    }

    public static EnumSet __odbGet_facets(Facets var0) {
        if (var0.__odbTracker != null) {
            var0.__odbTracker.beforeAccess(0);
        }

        return (EnumSet)var0.facets;
    }

    public static void __odbSet_dataObjectFacets(Facets var0, int var1) {
        if (var0.__odbTracker != null &&
           var0.__odbTracker.beforeModifyMember(1)) {
            var0.dataObjectFacets = var1;
            var0.__odbTracker.afterModify();
        } else {
            var0.dataObjectFacets = var1;
        }

    }

    public static int __odbGet_dataObjectFacets(Facets var0) {
        if (var0.__odbTracker != null) {
            var0.__odbTracker.beforeAccess(1);
        }

        return var0.dataObjectFacets;
    }

    public static int __odbGetMemberCount() {
        return 2;
    }

    public void __odbClear() {
        this.dataObjectFacets = 0;
    }

    public void __odbWriteContent(OType[] var1, OWriter var2) {
        var2.writeSInt32Value(this.dataObjectFacets);
    }

    public void __odbReadContent(OMember[] var1, OReader var2) {
        this.dataObjectFacets = var2.readSInt32Value();
    }

    public void __odbWriteMember(OMember var1, OWriter var2) {
        switch(var1.getMemberIx()) {
        case 0:
        default:
            throw new IllegalArgumentException();
        case 1:
            var2.writeSInt32Value(this.dataObjectFacets);
        }
    }
```

```
public void __odbReadMember(OMember var1, OReader var2) {
    switch(var1.getMemberIx()) {
    case 0:
    default:
        throw new IllegalArgumentException();
    case 1:
        this.dataObjectFacets = var2.readSInt32Value();
    }
}

public void __odbSetMember(int var1, Object var2) {
    switch(var1) {
    case 0:
        this.facets = (EnumSet)var2;
        return;
    case 1:
        this.dataObjectFacets = (Integer)var2;
        return;
    default:
        throw new IllegalArgumentException();
    }
}

public Object __odbGetMember(int var1) {
    switch(var1) {
    case 0:
        return this.facets;
    case 1:
        return this.dataObjectFacets;
    default:
        throw new IllegalArgumentException();
    }
}

public void __odbSetNumMember(int var1, long var2) {
    throw new IllegalArgumentException();
}

public long __odbGetNumMember(int var1) {
    switch(var1) {
    case 0:
    default:
        throw new IllegalArgumentException();
    case 1:
        return (long)this.dataObjectFacets;
    }
}

public void __odbClearMember(int var1) {
    switch(var1) {
    case 0:
        this.facets = null;
        return;
```

```
        case 1:
            this.dataObjectFacets = 0;
            return;
        default:
            throw new IllegalArgumentException();
        }
    }

    public long __odbGetVersionCrc() {
        return -8143500072684070832L;
    }

    public Object __odbsuperClone() throws CloneNotSupportedException {
        Facets var1 = (Facets)Tracker.fixClone(super.clone());
        return var1;
    }
}
```

Listing 16 – Classe Facets modifier par le enhancer ObjectDB

```
ConcreteModel mynet = workspace.getMainModel().createModel(petri);

ConcreteObject place1 = mynet.createObject(place);
place1.createProperty(count, 4L, mynet);
place1.createProperty(name, "P1", mynet);

ConcreteObject place2 = mynet.createObject(place);
place2.createProperty(count, 4L, mynet);
place2.createProperty(name, "P2", mynet);

ConcreteObject place3 = mynet.createObject(place);
place3.createProperty(count, 9L, mynet);
place3.createProperty(name, "P3", mynet);

ConcreteObject place4 = mynet.createObject(place);
place4.createProperty(count, 1L, mynet);
place4.createProperty(name, "P4", mynet);

ConcreteObject place5 = mynet.createObject(place);
place5.createProperty(count, 6L, mynet);
place5.createProperty(name, "P5", mynet);

ConcreteObject place6 = mynet.createObject(place);
place6.createProperty(count, 3L, mynet);
place6.createProperty(name, "P6", mynet);

ConcreteObject place7 = mynet.createObject(place);
place7.createProperty(count, 1L, mynet);
place7.createProperty(name, "P7", mynet);

ConcreteObject place8 = mynet.createObject(place);
place8.createProperty(count, 7L, mynet);
place8.createProperty(name, "P8", mynet);

ConcreteObject place9 = mynet.createObject(place);
place9.createProperty(count, 9L, mynet);
place9.createProperty(name, "P9", mynet);

ConcreteObject place10 = mynet.createObject(place);
place10.createProperty(count, 3L, mynet);
place10.createProperty(name, "P10", mynet);

ConcreteObject place11 = mynet.createObject(place);
place11.createProperty(count, 0L, mynet);
place11.createProperty(name, "P11", mynet);

ConcreteObject place12 = mynet.createObject(place);
place12.createProperty(count, 5L, mynet);
place12.createProperty(name, "P12", mynet);

ConcreteObject transition1 = mynet.createObject(transition);
mynet.createRole(source, place1, transition1);
mynet.createRole(source, place7, transition1);
```

```
mynet.createRole(target, transition1, place3);
mynet.createRole(target, transition1, place4);
mynet.createRole(target, transition1, place12);

ConcreteObject transition2 = mynet.createObject(transition);
mynet.createRole(source, place2, transition2);
mynet.createRole(source, place8, transition2);
mynet.createRole(source, place9, transition2);
mynet.createRole(target, transition2, place1);

ConcreteObject transition3 = mynet.createObject(transition);
mynet.createRole(source, place1, transition3);
mynet.createRole(source, place2, transition3);
mynet.createRole(target, transition3, place3);
mynet.createRole(target, transition3, place9);
mynet.createRole(target, transition3, place10);
mynet.createRole(target, transition3, place11);

ConcreteObject transition4 = mynet.createObject(transition);
mynet.createRole(source, place1, transition4);
mynet.createRole(target, transition4, place3);
mynet.createRole(target, transition4, place5);
mynet.createRole(target, transition4, place6);

ConcreteObject transition5 = mynet.createObject(transition);
mynet.createRole(source, place4, transition5);
mynet.createRole(source, place5, transition5);
mynet.createRole(target, transition5, place2);

ConcreteObject transition6 = mynet.createObject(transition);
mynet.createRole(source, place2, transition6);
mynet.createRole(target, transition6, place1);

ConcreteObject transition7 = mynet.createObject(transition);
mynet.createRole(source, place1, transition7);
mynet.createRole(target, transition7, place4);

ConcreteObject transition8 = mynet.createObject(transition);
mynet.createRole(source, place2, transition8);
mynet.createRole(source, place3, transition8);
mynet.createRole(target, transition8, place1);
```

Listing 17 – Exemple de création d'un modèle généré à l'aide de l'outil réalisé

Bibliographie

[Alz16] Hibatullah Alzahrani. Evolution of object-oriented database system. In *Global Journal of Computer Science and Technology : Computer Software & Data Engineering Volume 16 Issue 3*, 2016.

[Ban88] François Bancilhon. Object-oriented database systems. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, page 152–162, New York, NY, USA, 1988. Association for Computing Machinery.

[BCJM10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco : A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 173–174, New York, NY, USA, 2010. Association for Computing Machinery.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[BGS+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, a scalable persistence layer for emf models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications - Volume 8569*, pages 230–241, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[BK91] Rajiv D. Banker and Robert J. Kauffman. Reuse and productivity in integrated computer-aided software engineering : An empirical study. *MIS Q.*, 15(3) :375–401, October 1991.

[BK12] Konstantinos Barmpis and Dimitrios S. Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, New York, NY, USA, 2012. ACM.

[CFB10] Olivier Curé, David Célestin Faye, and Guillaume Blin. RIBStore : Data Management of RDF triples with RDFS Inferences. Preprint submited to a conference, April 2010.

[CR88] E. J. Chikofsky and B. L. Rubenstein. Case : reliability engineering for information systems. *IEEE Software*, 5(2) :11–16, 1988.

[Dan16] Gwendal Daniel. Efficient persistence and query techniques for very large models. In *SRC@MoDELS*, 2016.

[DSC16] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Prefetchml : A framework for prefetching and caching models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 318–328, New York, NY, USA, 2016. ACM.

[EH07] Vincent Englebert and Patrick Heymans. Towards more extensible metacase tools. In John Krogstie, Andreas Opdahl, and Guttorm Sindre, editors, *Advanced Information Systems Engineering*, pages 454–468, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Eng13] Vincent Englebert. Metadone : outil de support à l'ingénierie des langages de modélisation spécifique, February 2013.

[Eng20] Vincent Englebert. Syllabus d'ingénierie d'usines à logiciels, 2020.

[ESU97] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-case in practice : A case for kogge. In Antoni Olivé and Joan Antoni Pastor, editors, *Advanced Information Systems Engineering*, pages 203–216, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Gra] Grabats. 5th international workshop on graph-based tools. `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009`. Online ; accessed : 2020-04-28.

[GTSC15] Abel Gómez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot. Map-based transparent persistence for very large models. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering - Volume 9033*, pages 19–34, Berlin, Heidelberg, 2015. Springer-Verlag.

[Isa97] Hosein Isazadeh. Architectural analysis of metacase. 1997.

[KH10] Maximilian Koegel and Jonas Helming. Emfstore : A model repository for emf models. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 307–308, New York, NY, USA, 2010. ACM.

[Kim93] Won Kim. Object-oriented database systems : Promises, reality, and future. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, page 676–687, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[LMB+01] Akos Ledeczi, M Maroti, A Bakay, Gabor Karsai, J Garrett, C Thomason, G Nordstrom, J Sprinkle, and Péter Völgyesi. The generic modeling environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary*, 17, 01 2001.

[LRH06] L. Li, Y. Ru, and C. N. Hadjicostis. Least-cost transition firing sequence estimation in labeled petri nets. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 416–421, 2006.

[Met] MetaCase. Abc to metacase technology. `https://www.metacase.com/papers/ABC_to_metaCASE.pdf`.

[OS97] Andreas L. Opdahl and Guttorm Sindre. Facet modelling : An approach to flexible and integrated conceptual modelling. *Information Systems*, 22(5) :291 – 323, 1997.

[PCM11] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa : A scalable approach for persisting and accessing large models. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag.

[PK88] D. E. Perry and G. E. Kaiser. Models of software development environments. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, page 60–68, Washington, DC, USA, 1988. IEEE Computer Society Press.

[Qat09] Hazem Kathem Qattous. Constraint specification by example in a meta-case tool. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, page 13–16, New York, NY, USA, 2009. Association for Computing Machinery.

[Rub10] Daniel Rubio. Optimizing jpa performance : An eclipselink, hibernate, and openjpa comparison. `http://dzone.com/articles/jpa-performance-optimization`, 2010. Online ; accessed 25-July-2020.

[SBG08] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervals. Collaborative software engineering on large-scale models : Requirements and experience in modelbus. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 674–681, New York, NY, USA, 2008. ACM.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled : A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[Sif77] Joseph Sifakis. Use of petri nets for performance evaluation. In *Proceedings of the Third International Symposium on Measuring, Modelling and Evaluating Computer Systems*, page 75–93, NLD, 1977. North-Holland Publishing Co.

[Sof12] ObjectDB Software. Jpa performance benchmark. `http://www.jpab.org/All/All/All.htm`, 2012. Online ; accessed 25-July-2020.

[SZFK12] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 102–118, Berlin, Heidelberg, 2012. Springer-Verlag.

[TR03] Juha-pekka Tolvanen and Matti Rossi. Metaedit+ : Defining and using domain-specific modeling languages and code generators. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 92–93, 01 2003.

[Wik20a] Wikipedia. Eclipse Modeling Framework Wikipedia. `http://fr.wikipedia.org/w/index.php?title=Eclipse%20Modeling%20Framework&oldid=157736351`, 2020. Online ; accessed 18-July-2020.

[Wik20b] Wikipedia. OSGI Wikipedia. `http://en.wikipedia.org/wiki/OSGi`, 2020. Online ; accessed 20-July-2020.

[Wik20c] Wikipedia. Scalability Wikipedia. `http://fr.wikipedia.org/wiki/Scalability`, 2020. Online ; accessed 20-July-2020.

[Win12] M. Winand. *SQL Performance Explained : Everything Developers Need to Know about SQL Performance*. M. Winand, 2012.